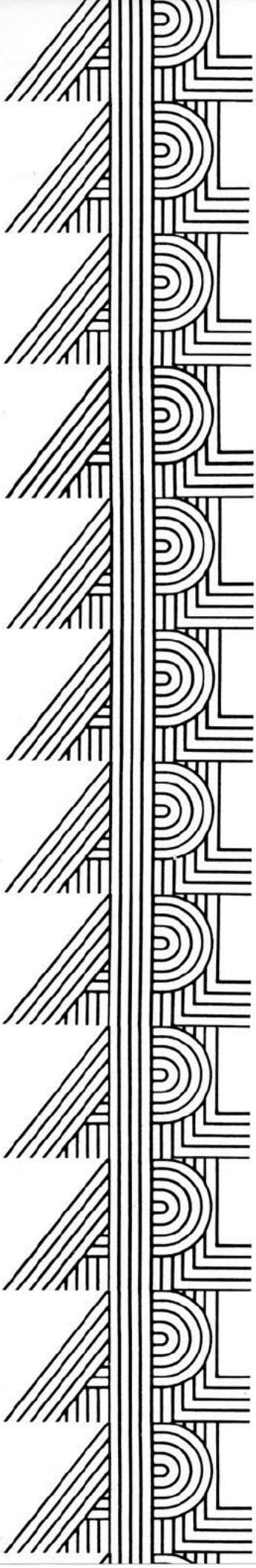


SHARP APL REFERENCE MANUAL

REUTERS

SECOND EDITION

No part of this manual may be copied or reproduced, transmitted or distributed in any form or by any means without the prior written consent of the copyright holder.

For any information or communication about this reference manual, please contact:

APL Software Division
Reuters Information Services (Canada) Limited
Suite 1900, Exchange Tower
2 First Canadian Place
Toronto, Ontario, Canada M5X 1E3

Information in this reference manual is subject to change without notice. The APL Software Division, its agents and employees shall not be liable for any loss whatsoever resulting from errors or omissions in this reference manual.

© 1991 APL Software Division, Reuters Information Services (Canada) Limited
All rights reserved
Printed in Canada

Publication code: 1390-9112-E1

Contents

About this Manual 0–1

Related Publications 0–1

Conventions Used in this Manual 0–1

What You Should Not Expect from this Manual 0–2

1. Overview 1–1

APL: Visible and Invisible 1–1

Introduction by Example 1–2
Arrays 1–5

Origins of APL 1–6
Continuing Divergent and Convergent Development 1–6
Development Related to Array Processing 1–7
Interpretive Basis for APL Systems 1–8
Access to Files 1–9
Shared Work and Private Work 1–9

Summary of Features of SHARP APL 1–9
Features Beyond Those Required by the ISO Standard 1–9
Features Added Since the Last Edition 1–11
Features Changed or Discontinued 1–15

2. Tasks, Users, Libraries 2–1

Timesharing 2–1

Users 2–1
Users: Ordinary, Privileged, Constrained, and Profiled 2–2

Tasks 2–3
Types of Tasks 2–3
Every Task Has Its Own Active Workspace 2–4
Saved Workspaces 2–4
Size of the Active Workspace 2–5
Ways to Start a Task 2–5
Duration of a Task 2–5
Work Saved at Forced Termination, Line Drop, or Crash 2–6
Decision to `)load continue` 2–7
A Task's Access to Saved Workspaces 2–8
A Task's Access to Files 2–9

Starting a T-Task 2–10
Sign-on 2–10
Automatic Workspace Load 2–12
Immediate Execution 2–13
Input Modes 2–13

Starting an S-Task 2–14

Starting an Immediate N-Task 2–15

Starting a B-Task or a Scheduled N-Task 2–16
An N-Task or B-Task Cannot Obtain Interactive Input 2–16

Tasks Belonging to Constrained Users 2–16

3. Grammar 3–1

APL Syntax 3–1
Syntactic Classes 3–1
Evaluation 3–3
Order of Execution 3–3
Ruthlessly Consistent Order of Execution 3–4
Infix Notation 3–4
Ambivalence 3–4
Niladic, Monadic, and Dyadic Use 3–5
Grouping 3–5
Tree Structure of a Sentence 3–6
Left-to-Right and Leaf-to-Root 3–7
Precedence of Verbs, Adverbs, and Conjunctions 3–7
Precedence in a Sentence with Adverbs or Conjunctions 3–8

Other Mechanisms of Grouping 3–9
Brackets Group in the Same Way as Parentheses 3–9
Semicolon Is a Separator 3–10
Other Punctuation 3–11
Colon and Label 3–11
Lamp and Comment 3–12
Arrays 3–12
Types of Nouns 3–13
No Declarations 3–13

Analysis of an Input Line 3–13
Preliminary Checks for) and ∇ 3–14
Scanning the Line for Quotes and Comments 3–14
Dividing the Line into Sentences 3–15
Types of Sentences 3–15
Dividing a Sentence into Words 3–16
Multi-character Words 3–16
Blanks 3–16
Scanning Strategy for Locating Words 3–17
Numeric Constants 3–17
Complex and Exponential Number Formats 3–18
List Constants 3–18
Names 3–19
Assigning Words in a Sentence to Syntactic Classes 3–20

4. Nouns and Pronouns 4–1

Rank and Shape 4–1
Reporting Shape and Rank 4–2
Arrays and Cells 4–2
Complementary Partition 4–4
Major Cells 4–5
Infinite Rank 4–5

Types of Noun Items 4–5
Homogeneity of Arrays 4–5
Empty Arrays 4–5
Numbers 4–6
Internal Representations of Numbers 4–6
Character Data 4–8
APL Character Set 4–8
Boxed Arrays 4–9
Partitioning 4–10

Display of Boxes 4–11
Packages 4–11

Input and Output 4–12
Constants 4–12
Creating Higher-Rank Arrays 4–14
Creating the Value of a Box 4–14

Display of Nouns 4–15
Position and Spacing in an Array of Boxes 4–15
Separating Successive Columns of a Numeric Array 4–16
Separating Successive Planes of Higher-Rank Arrays 4–16
Folding Wide Displays 4–17
Folding a Character Array 4–17
Folding a Numeric List 4–17
Folding a Numeric Table or Higher-Rank Array 4–18
Folding an Array of Boxes 4–18
Display of an Empty Array 4–18

Interface Nouns: □ and ⍞ 4–19
Evaluated Input 4–19
Input in Response to □ 4–20
Character Input 4–21
Escape from ⍞-Input and □-Input 4–21
Explicit Output from □ and ⍞ 4–22
Folding of ⍞ Output 4–22
Input on the Same Line as the Character Prompt 4–23
Discarding the Prompt 4–23

5. Naming Nouns and Verbs 5–1

←: Copula 5–1

Workspace: The Universe of Names 5–1

Localization of Names 5–2

Reassignment and Name Conflicts 5–3

Names for User-Defined Objects 5–3

Indexed Assignment 5–4

The Result of Assignment 5–5

Multiple Assignment 5–5
Sentences Producing No Display 5–5

Other Mechanisms that Produce Named Objects 5–6

6. Verbs 6–1

Default Argument Ranks 6–1

Agreement 6–2

Scalar Verbs 6–2

Extending the Empty Frame 6–3

Unbounded Rank 6–3

Unspecified Rank 6–3

Reduction and Identity Elements 6–4

Result Rank and Shape 6–7

Verb Definitions 6–8

+ − × ÷: **Plus, Minus, Times, Divide** 6–8
Dyads + − × ÷ 6–8
Cartesian Representation of Complex Product 6–8
Polar Representation of Complex Product 6–9
Division by Zero 6–9
Monad + ... 6–9
Monad − 6–10
Monad × 6–10
Monad ÷ 6–10

⌹: Matrix Inverse; Matrix Divide 6–11
Monad ⌹ 6–11
Dyad ⌹ 6–11
Frame and Cells with Dyad ⌹ 6–13

⋆ ⊛: Power, Log 6–14
Monads ⋆ and ⊛ 6–14
Dyad ⋆ 6–15
Dyad ⊛ 6–16

○: Circle Verbs 6–17
Monad ○ 6–17
Dyad ○ 6–17
Selection of Function by Left Argument of ○ 6–18
Geometric View of the Pythagorean and Circular Functions 6–18
Geometric View of the Hyperbolic Functions 6–19
Representations of Complex Numbers 6–21

⌈ ⌊: Floor, Ceiling; Max, Min 6–22
Monads ⌊ and ⌈ 6–22
Tolerant Floor and Ceiling 6–23
Floor of a Complex Number 6–23
Dyads ⌊ and ⌈ 6–25

|: Magnitude; Residue 6–26
Monad | 6–26
Dyad | 6–26

!: Factorial; Combinations 6–28
Monad ! 6–28
Dyad ! 6–29

?: Roll; Deal 6–30
Monad ? 6–30
Dyad ? 6–30
Argument Rank 6–30
Pseudo-random Algorithm and ⎕rl 6–31

< > ⊃: Box, Open; Link 6–32
Monad > 6–33
Tolerant Open 6–33
Fill Elements 6–34

Monad ⊃ 6–35
Dyad ⊃ 6–35

< >: Less; Greater 6–37

≤: Less than or Equal 6–37

≥: Greater than or Equal 6–37

= ≠: Nubsieve; Equal, Unequal 6–38
Monad ≠ 6–38
Dyad = ... 6–40
Tolerant Comparison of Real Numbers 6–40
Dyad ≠ 6–41

≡: Match 6–41

~: Not 6–42

∧ ∨ ⍲ ⍱: LCM, GCD; And, Or; Nand, Nor 6–42
Boolean Arguments 6–42
Non-Boolean Arguments 6–43
Tolerant GCD and LCM 6–44

⊢ ⊣: Right, Left; Dex, Lev 6–44
Dyads ⊢ and ⊣ 6–44

⊤ ⊥: Encode; Decode 6–46
Dyad ⊤ 6–46
Relation Between Representation and Residue 6–46
Higher-Rank Arguments to ⊤ 6–48
Forcing Representation Cells to the Last Axis 6–49
Dyad ⊥ 6–50
Item Left Argument 6–50
Higher-Rank Right Argument 6–51
Higher-Rank Left Argument 6–51
Length-1 Base 6–51
Forcing Representation Cells to the Last Axis 6–52
Limitations on Inverse 6–52
Evaluation of Polynomials 6–53

⍕: Format 6–53

Default Display 6–53
Monad ⍕ 6–54
Array-wide Format vs. Item-by-Item Format 6–55
Choice of Fixed-Point or Exponential Format for Real Numbers 6–55
Mixed Formats in Representation of a List 6–56
Appearance of Fixed-Point Format 6–56
Appearance of Exponential Format 6–57
Appearance of Complex Format 6–58
Field Width in Tables 6–58
Blanks Between Columns 6–59
Examples of Monad ⍕ 6–59
Separation Between Planes of Higher-Rank Arrays 6–60
Format of an Array of Boxes 6–60
Effect of □*ps* on Position and Spacing of Boxes 6–61
Position: The First Two Elements of □*ps* 6–62
Spacing: The Last Two Elements of □*ps* 6–62
Dyad ⍕ 6–63
Dyad ⍕ with Empty α 6–64
Dyad ⍕ with Numeric α 6–64
Fixed-Point Representation 6–64
Exponential Representation 6–66
Dyad ⍕ with Character α 6–67
Numeric Fields within the Pattern 6–67
Decorators within the Pattern 6–68
Treatment of the Decimal Point and Decimal Digits 6–69
Treatment of the Sign 6–69
Alternative Conventions Regarding Decimal Point and Grouping 6–70
Dot and Comma 6–71
Significance of the Control Digits 0 to 9 6–72
Effect of □*fc* on ⍕ Format by Example 6–77

⍎ : Execute 6–78
Result of ⍎⍵ 6–79
Display of Result 6–79
Evaluation of Multiple Sentences 6–80
Occurrence in the State Indicator 6–80
Branch within Execute 6–80
Errors Encountered During Execution 6–81
□*signal* within Execute 6–81

⍪ , : Table, Ravel; Join 6–82
Monads , and ⍪ 6–82
Dyads , and ⍪ 6–85
Restrictions on the Domains of ⍪ or , ... 6–85
Shape of Arguments and Result 6–85

Joining Two Arrays Along a New Axis 6–87
Joined Arrays May Differ in Rank by 1 6–90
Joining an Item to an Array 6–90
Joining Modified by Bracket-Axis Notation 6–91

⊖ ⌽: **Reverse, Upset; Rotate, Rowel** 6–92
Monads ⊖ and ⌽ 6–92
Dyads ⌽ and ⊖ 6–93
⌽ and ⊖ Modified by Bracket-Axis 6–95
Upset and Rowel in Terms of Rotate and Reverse 6–95

⍉: **Transpose; Cant** 6–96
Monad ⍉ 6–96
Dyad ⍉ 6–97
Transpositions that Retain All the Axes 6–98
Diagonal Slices 6–101

↑ ↓: **Raze; Take, Drop** 6–105
Monad ↓ 6–105
Raze of Empty Arrays 6–106
Raze Along Unit Axis 6–106
Dyad ↑ and ↓ 6–106
Take with Short Left Argument 6–108
Drop with Short Left Argument 6–109
Dropping More Elements than Exist 6–109
Over-Take and Padding 6–109
Take When the Right Argument Is Empty 6–110
Take and Drop Applied to Item ⍺ 6–110
Take and Drop Modified by the Rank Conjunction 6–111

⍋ ⍒: **Grade up, Grade down** 6–112
Monads ⍋ and ⍒ 6–112
Grade is Not Tolerant 6–113
Dyads ⍋ or ⍒ Grade Character Arrays 6–113
Collating Sequence with Tied Weights 6–114
Characters with Identical Weights 6–116

⍴: **Shape; Reshape** 6–117
Monad ⍴ 6–117
Dyad ⍴ 6–118

∈: **Member** 6–120
Membership Modified by the Rank Conjunction 6–121

∈ : Find, In 6–122
Dyad ∈ 6–122

⍳ : Iota; Count, Indexof 6–123
Monad ⍳ 6–124
Default Rank of Monad ⍳ 6–124
Dyad ⍳ 6–124
Tolerant Comparison in Dyad ⍳ 6–125

⍺[⍵] : Bracket Indexing 6–126
Shape of the Result Produced by Bracket Indexing 6–127
Keeping All Positions Along an Axis 6–127

Indexed Assignment 6–129
Effect of Indexed Assignment 6–130
Explicit Result of Indexed Assignment 6–131

7. Adverbs and Conjunctions 7–1

Precedence of Adverbs and Conjunctions 7–1

Valence of Adverbs and Conjunctions 7–2

Syntactic Class of the Arguments to an Adverb or Conjunction 7–2

Ambivalence of the Derived Verb 7–2

Consecutive Adverbs and Conjunctions 7–2

Example to Illustrate Order of Evaluation 7–3

Look-Ahead 7–4

Separating Side-by-Side Constants 7–4

Adverbs 7–6

***v*⌿⍵ *v*/⍵ *n*/⍵ *n*⌿⍵: Reduction, Replicate, Compress** 7–6

v⍀⍵ *v*\⍵ *n*⍀⍵ *n*\⍵: **Scan, Expand** 7–6
Summary of Forms 7–6
Monads Scan and Reduce 7–7
Effect of Order of Grouping on Scan and Reduce 7–7
Reduce When Argument Has One Cell or None 7–8
Some Examples of Reduce 7–10
Some Examples of Scan 7–10
The Derived Verbs Replicate and Expand 7–12
Replicate 7–12
Compress 7–13
Expand: Introducing "Fill" Cells 7–14
Derived Verbs Modified by Axis Bracket Notation 7–14

Conjunctions . ⍤ ⍥ ⍣ 7–15
Grouping in the Discussion of Conjunctions 7–15
Conjunctions ⍤ ⍥ ⍣ with Verb Arguments 7–16
Under Requires a Known Inverse 7–16
Derived Verbs Produced by Composition 7–17
Default Rank of the Derived Verb 7–17
The Rank Conjunction *v*⍤*n* 7–19
Argument Rank vs. Complementary Rank 7–20
Within Cells, the "Ordinary" Rank Applies 7–20
Default Rank Not Defined for Some Verbs 7–20
Examples of Rank 7–21
The Cut Conjunction *n*⍤*v* 7–22
Types of Cuts 7–22
Cells Selected by Table 7–23
Cells Selected by Boolean Partition 7–24
Monadic Use of 1– and 2–Cuts 7–26
Shape of the Result Produced by Cut 7–26
Dot Conjunctions 7–26
Tie and Outer Product 7–27
Inner Product 7–29
Result Shape of Inner Product 7–29
Extending One Argument to Match the Other's Length 7–30
Inner Product and Matrix Product 7–30
Monadic Use: The Alternant 7–33
Non-Square Arguments 7–34

8. Verb Formation 8–1

Mechanisms for Defining and Editing Verbs 8–1

Representation of a User-Defined Verb 8–3
Canonical Form 8–4
Names in the Header 8–5
Headers for Monadic, Dyadic, and Ambivalent Verbs 8–5
Localization of Names 8–6

Using the ∇-Editor 8–6
Prompts in the ∇-Editor 8–7
Editing Commands 8–8
Interpolating Lines Between Existing Lines 8–9
Empty Lines 8–10
Syntactically Invalid Lines 8–10
Box-drawing Characters 8–10
Editing Occurs in Real Time 8–10
Closing the Definition 8–10
Locking the Definition 8–11
Editing Can Prevent Resumption of Execution 8–11
Distinguishing Active, Pendent, and Suspended Verbs 8–11
Damage to the Active Verb 8–12
Damage to a Pendent Verb 8–13

9. Control of Execution 9–1

Programming and Immediate Execution 9–1
Status of Work in Progress 9–2
Suspension Due to Error or Interrupt 9–3
Pendent vs. Suspended Verbs 9–4
The Line Counter 9–4
Local Names at Each Level of the State Indicator 9–5
Dynamic Localization 9–6
Steps at the Start of Execution of a User-Defined Verb 9–7
Example of Argument and Result Passing 9–7
Multiple Copies of Arrays Share the Same Space 9–8
Value Returned by the Verb at Completion 9–9
Branch Controls the Sequence in Which Lines Are Executed 9–9
Form of a Branch Statement 9–9
Result and Effect of a Branch 9–10
Effect When Destination Is Zero or Out of Range 9–10
Effect When Destination Is Empty 9–10
Branch in Execute 9–11
Branch in a Line with Diamonds 9–11
Branch to Resume Suspended Execution 9–11
Resumption Following a Mid-line Halt 9–12

Naked Branch to Abandon Execution 9–12
Clearing the Entire State Indicator 9–12
Branch During □-Input 9–12
Calculating the Destination of a Branch 9–13
Conditional Branch 9–13
Conditional Execution of Statements on the Same Line 9–13
Multi-way Branch 9–13
Branch with Counter 9–14

Stop and Trace Controls 9–14
Effects of Stop 9–15
Effects of Trace 9–15
Effect of Editing on Trace or Display 9–16
Effect of Storing and Retrieving a Verb with Stop or Trace Set 9–16

10. Event Handling 10–1

Errors and Interrupts 10–1
Classifying Interrupts 10–2
Weak and Strong Interrupts 10–2
Strong Interrupts and Restartability 10–2
Signaled Errors 10–3
Trap, Action, and Recovery 10–4
Fields within Each Trap Statement 10–4
Unreportable or Untrappable Events 10–6
Environment Condition Prohibits Suspension 10–6
Effect of `1` as the Value of `□ec` 10–7
Values of `□ec` 10–8
`□ec←1` in the SI Stack Prohibits Suspension 10–8
Response to an Event When Stop is Prohibited 10–9
What Happens When an Event Occurs 10–10
Caution: Trapping Interrupts or Prohibiting Suspension 10–11
Caution: After Suspension, `□trap` Is Still Active 10–12

11. Shared Variables 11–1

A Discipline of Communication 11–1

The Shared Variable Processor 11–2
Sharing with Non-APL Programs 11–2
Auxiliary Processors 11–3

How Name-Sharing Works 11–3
Sending and Receiving 11–4
Set, Use, and Reference 11–4

Establishing a Shared Name 11–5
Offers and Retractions 11–6
Surrogate Names 11–6
Processor ID, Clone ID, and System ID 11–7
Degree of Coupling 11–7
Duration of Sharing 11–8
Localization and Shared Names 11–8
Inquiring About Offers Made to You by Others 11–9
The Shared Name's State and Controls 11–9
Point of View in □svs and □svc 11–9
First Pair: Who Has Set a New Value? 11–10
Second Pair: Who Is Aware of the Current Value? 11–10
Possible Values of □svs 11–10
Access Control 11–11
Setting the Shared Variable Control 11–12
Effective Access Control 11–13
Common Values of the Effective Access Control 11–13
Verifying that Access Control Has Been Set 11–15
Graphic Representation of Communication Control 11–15
Representation of Access Control 11–17
Exceptional Routes 11–17
Attempting to Carry Out an Action That Is Inhibited 11–18
Shared Names and □pack, 4 □ws and 6 □ws 11–19
Which Actions Will Not Be Delayed? 11–19
Waiting for a State Change 11–19
What Counts as a State Change? 11–20
Sequence for Using □sc 11–21
Saving a Workspace Containing Shared Variables 11–21

12. S-Tasks 12–1

Starting an S-Task 12–2
Duration of an S-Task 12–2
AP1 and Controlling Task Communication 12–3
Prefix Codes 12–4
Prefix Codes Sent to AP1 12–4
Prefix Codes Sent from AP1 12–5
Segmented Transmissions from AP1 12–5
Interpretation of Prefix Codes 12–6

Binary Representation of `prefix[3]` 12–6
Common Values of `prefix[3]` 12–7
Freestanding S-Task 12–7
Default Value of Independence Bit 12–8
Reconnecting to a Freestanding S-Task 12–8
Sample Programs Useful in Debugging S-Tasks 12–9
Sample Programs for a Server for Multiple Users 12–11

13. Batch Tasks 13–1

How Batch Tasks Are Started and Controlled 13–1
Alternative Starting Points for an N-Task 13–2

The Owner of a Batch Task 13–3

Limits on Type and Number of Tasks 13–3

Restart After Interrupted Work 13–3

Pricing and Priorities 13–4

Starting an N- or B-Task 13–4
Sample Arguments for `□run` 13–6
Result of `□run` 13–7
Distinguishing the N-Task from Its Parent 13–7
Reporting on the Status of an N-Task 13–8

Submitting a B-Task Request 13–8

14. APL Component Files 14–1

Concepts Relevant to Files in General 14–1
Opening a File 14–1
Records 14–2
File Format 14–2
Access to Individual Records 14–2
Each Record is Interpreted According to the File's Format 14–2

Concepts of APL Component Files 14–3

Ownership 14–3
Organization into Libraries 14–3
Limits and Quotas 14–4
Components 14–4
Indexed Access 14–5
Component Information 14–5
Self-Describing Data 14–5
Variable-Size Components 14–5
Open-Ended Size 14–6
File Primitives 14–6
Tying Links a File to the Verbs that Use It 14–6
Duration of a File Tie 14–7
Concurrent Use 14–7
Shared Use 14–7
Library Access Controls 14–7
Controlled Library Operations 14–8
File Access Controls 14–10
Passnumbers and Restrictions on Access 14–11
Access Control by User Number and Verb 14–11
Permission Codes 14–11
Tying a File with a Passnumber 14–13
Effective Permission 14–14
Concurrency 14–15
Monadic Use of the Hold Verbs 14–16
Dyadic Use of the Hold Verbs 14–16
Ending a File Hold 14–17
Normal Use of File Hold 14–17

Summary of File Verbs 14–19
File Verbs: Inquiry 14–20
File Verbs: Existence 14–20
File Verbs: Linkage 14–20
File Verbs: Description and Limits 14–20
File Verbs: File and Library Access 14–21
File Verbs: Concurrency 14–21
File Verbs: Use 14–21

15. APL Characters 15–1

Distinctions Between Characters and Their Representations 15–1
Mappings Between Keys, Glyphs, and Characters 15–2

Characters in the APL System 15–2

Basic Alphabet 15–3
Numerals 15–3
Symbols 15–3
Graphics 15–3
National Use Characters 15–3
Control Characters 15–4
Normal and Reversed Alphabets 15–4
Historical Perspective on the Alphabet Change 15–5
Text Support for Non-APL Terminals 15–5
The Third Alphabet 15–6
Keyboard Alternatives 15–6
Maintaining Data Independence 15–7

Internal Representation of APL Characters 15–7
Translation for the Import and Export of Data 15–8

Transmission Codes and Overstrike Conventions 15–10

16. System Nouns and Verbs 16–1

Results of System Verbs 16–2

Alphabetical List of System Names 16–3

□*ai*: Accounting Information 16–3

□*append*: Append Noun to Tied File 16–4

□*appendr*: Append Noun to Tied File with Result 16–4

□*arbin*: Arbitrary Input with Arbitrary Output 16–5

□*arbout*: Arbitrary Output 16–5

□*av*: Atomic Vector 16–7

□*avail*: File System Availability 16–7

□*bounce*: Terminate Task 16–8

□*cr*: Canonical Representation 16–8

□*create*: Create a File 16–9

□*ct*: Comparison Tolerance 16–10
Verbs Affected by □*ct* 16–10

□*dl*: Delay Execution 16–12
Behavior of Delay When Interrupted 16–12

□*drop*: Drop File Components 16–13

□*ec*: Environment Condition 16–14

□*er*: Event Report 16–15

□*erase*: Erase a Tied File 16–17

□*ex*: Expunge Objects 16–17

□*fc*: Format Control 16–18

□*fd*: Verb Definition 16–19

□*fhold*: Hold Tied Files 16–20

□*hold*: Hold Tied Files 16–20
Monads □*fhold* and □*hold* 16–22
Dyad □*fhold* 16–22
Application to a Range of Numbers 16–23
Duration of a File Hold 16–23

□*fi*: Input Format Conversion 16–24
Use of □*fi* with □*vi* 16–25

□*fm*: Function Monitor 16–25
Dyad □*fm* 16–25
Monad □*fm* 16–27
Storage Allocated to Monitoring Data 16–27

Monitoring a Recursive Definition 16–28

□*fmt*: **Format Output** 16–29
Format Phrases 16–29
Repetition Factors 16–31
Type and Shape of the Right Argument of □*fmt* 16–31
Shape of the Result of □*fmt* 16–31
Qualifiers and Decorators 16–31
Examples of □*fmt* 16–34
Obsolescence of "Paren/Semicolon" Argument to □*fmt* 16–35

□*fx*: **Fix a Definition** 16–35

□*hold*: **Hold Tied Files** 16–36

□*ht*: **Horizontal Tab** 16–36

□*io*: **Index Origin** 16–36
Verbs Affected by □*io* 16–37

□*lc*: **Line Counter** 16–38

□*lib*: **File Library** 16–39

□*load*: **Load a Workspace** 16–39

□*qload*: **Load a Workspace Quietly** 16–39

□*lx*: **Latent Expression** 16–40
Example of □*lx* 16–40

□*names*: **Names of Tied Files** 16–42

□*nc*: **Name Class of Identifiers** 16–42

□*nl*: **Name List** 16–42

□*nums*: **Tie Numbers of Tied Files** 16–43

□*out*: **Record Session I/O to File** 16–43

Timing of □*out* Result 16–44
Control Messages 16–44
List of Control Messages 16–45

□*pack*: **Build a Package** 16–47

□*pdef*: **Package Define** 16–47

□*ppdef*: **Protective Package Define** 16–47
Protective Definition 16–48

□*pex*: **Package Expunge** 16–48

□*pins*: **Package Insert** 16–48

□*plock*: **Package Lock** 16–49

□*pnames*: **Package Names** 16–49

□*pnc*: **Package Name Class** 16–49

□*pp*: **Print Precision** 16–50

□*ppdef*: **Protective Package Define** 16–50

□*ps*: **Position and Spacing in Display** 16–51

□*psel*: **Package Select** 16–52

□*pval*: **Package Value** 16–52

□*pw*: **Page Width** 16–53

□*qload*: **Load a Workspace Quietly** 16–53

□*rdac*: **Read File Access Table** 16–53

□*rdacl*: **Read Library Access Table** 16–54

□*rdci*: Read File Component Information 16–55

□*rdfi*: Read File Timestamp Information 16–56

□*read*: Read Component of a Tied File 16–57

□*rename*: Rename a Tied File 16–58

□*replace*: Replace Noun Stored in a Component 16–59

□*resize*: Set File Size Limit 16–60

□*rl*: Random Link 16–61

□*run*: Start N-Task 16–62
Result Codes 16–63

□*runs*: Report Running Tasks 16–64

□*sc*: Shared Variable State Change 16–65
Setting a □*sc* Time-out 16–65
Impending Shutdown 16–65

□*signal*: Terminate and Signal Event 16–66

□*size*: Size of Tied File 16–67

□*sp*: Session Parameter 16–68

□*stac*: Set File Access Table 16–68

□*stacl*: Set Library Access Table 16–69

□*stie*: Share Tie 16–70

□*svc*: Shared Variable Control 16–72

□*svn*: Shared Variable Clone ID 16–72

□*svo*: **Shared Variable Offer** 16–74
Left Argument 16–74
Right Argument 16–74
Monad □*svo* 16–74

□*svq*: **Shared Variable Query** 16–75

□*svr*: **Shared Variable Retract** 16–76

□*svs*: **Shared Variable State** 16–76

□*tie*: **Exclusive Tie** 16–77

□*trap*: **Event Trap Control** 16–79
□*trap* Event Numbers Field 16–80
□*trap* Qualifier Field 16–80
□*trap* Action Field 16–80
□*trap* Recovery Expression 16–81
Action *s* : Stop 16–81
Action *n* : Next 16–81
Action *i* : Immediate 16–81
Action *e* : Execute 16–82
Action *c* : Cut Back and Execute 16–82
Action *d* : Do 16–82

□*ts*: **Timestamp** 16–83

□*twsid*: **Termination Workspace ID** 16–83

□*ul*: **User Load** 16–85

□*untie*: **Untie Tied Files** 16–85

□*vi*: **Verification of Input Format** 16–85

□*wa*: **Work Area Available** 16–86

□*ws*: **Workspace Information** 16–86

⌶: **I-Beam Verbs** 16–89

17. System Commands 17–1

) : Recall Previous APL Line 17–3

)*nnnnnnn:pass* : Sign-on to SHARP APL 17–4

)`blot`: Security Blot 17–4

)`cc`: Character Control 17–5

)`clear`: Clear Active Workspace 17–6
Conditions in a Clear Workspace 17–6

)`continue`: Terminate APL Session for Restart 17–7

)`copy`: Copy Objects into Workspace 17–8

)`pcopy`: Protected Copy into Workspace 17–8

)`cpulim`: CPU Limit 17–9

)`drop`: Delete a Workspace 17–10

)`erase`: Erase Global Objects 17–10

)`fns`: Display Verb Names 17–11

)`group`: Alter a Group 17–11

)`grp`: Display Group Member Names 17–12

)`grps`: Display Names of Groups 17–12

)`keyboard`: Alter Keyboard State 17–12

)`lib`: Display Saved Workspace Names 17–13

`)load`: **Activate a Workspace** 17–14
The `)xload` Variant 17–14

`)msg`: **Send Message with Reply** 17–15

`)msgn`: **Send Message with No Reply** 17–15

`)opr`: **Send Message to Operator** 17–15

`)oprn`: **Send Message to Operator with No Reply** 17–15
Message Delivery 17–15
Message to Which a Reply Is Expected 17–16

`)nlt`: **Set Language** 17–17

`)off`: **Terminate APL Session** 17–18

`)opr`: **Send Message to Operator** 17–18

`)oprn`: **Send Message to Operator with No Reply** 17–18

`)pcopy`: **Protected Object Copy** 17–18

`)port`: **Locate User** 17–19

`)reset`: **Clear SI Stack** 17–19

`)save`: **Save Active Workspace** 17–20
Workspace Locks 17–20

`)seal`: **Seal Workspace** 17–21

`)sesm`: **Session Handler Commands** 17–21

`)sic` : **State Indicator Clear** 17–22

`)si`: **State Indicator** 17–22

`)sinl`: **State Indicator with Namelist** 17–22

`)siv`: **State Indicator with Namelist** 17–22

`)symbols`: **Set Symbol Table Size** 17–24

`)terminal`: **Terminal Type** 17–24

`)vars`: **Display Noun Names** 17–25

`)wsid`: **Set Workspace Name** 17–26

`)xload`: **Load Workspace without Autostart** 17–26

18. Messages 18–1

Sources of messages and trouble reports 18–1

Trappable Errors and Interrupts 18–2

Form of an Error Message 18–2
How Much Has Been Executed 18–3
Point of View in Describing an Error 18–3

National Language Translation: `)nlt` 18–3

Error Messages By Event Number 18–5
System Nouns with Invalid Values 18–14

Interrupt Events 18–15

Alphabetic List of Messages 18–17

Figures

Figure 1–1 : Elementary examples to illustrate some aspects of APL.....1–2

Figure 2–1 : Sign-on of an interactive task.....2–11

Figure 2–2 : Comparison of User Types.....2–17

Figure 3–1 : Tree representation of `a×b`.....3–6

Figure 3–2 : Tree representation of `(a+b)×(a−b)`.....3–6

Figure 3–3 : Characters permitted in names and numeric constants.....3–20

Figure 3–4 : Symbols in syntactic classes.....3–20

Figure 4–1 : Alternative ways of partitioning a 2-by-3-by-4 array into cells.....4–4

Figure 4–2 : A 2-by-3-by-4 array of boxes.....4–10

Figure 6–1 : Default argument ranks of primitive verbs.....6–3

Figure 6–2 : Identity elements for scalar dyads.....6–6

Figure 6–3 : Shape of the argument and result of dyad ⌹.....6–14

Figure 6–4 : Pythagorean and circular verbs in terms of the unit circle.....6–19

Figure 6–5 : Hyperbolic verbs in terms of the unit hyperbola.....6–20

Figure 6–6 : Mapping of points in a unit square to complex integers.....6–24

Figure 6–7 : The complex plane tiled by areas having equal floors.....6–25

Figure 6–8 : Schematic representation of `3 ¯3|10` *and* `3 ¯3|¯8`.....6–27

Figure 6–9: Shape of result cells produced by tolerant open.....6–34

Figure 6–10: Preservation of grid structure in formatting an array of boxes.....6–60

Figure 6–11: Effect of monad ⍪ at ranks `5` *through* `1`.....6–83

Figure 6–12: Dyad ⍪ rank 4, 3, and 2.....6–87

Figure 6–13: Dyad ⍪ rank 1.....6–87

Figure 6–14: Reversing the order of a table's last-axis cells.....6–93

Figure 6–15: Reversing the order of last-axis cells in a rank-3 array.....6–93

Figure 6–16: Axes of argument and axes of the result.....6–98

Figure 6–17: Dyadic transpose of a rank-4 array.....6–100

Figure 6–18: Visualization of a rank-`4` *array and its* `3 1 4 2` *transpose*.....6–101

Figure 6–19: Axes mapped together during transpose.....6–102

Figure 6–20: Visualization of `1 3 2 3` *transpose*.....6–102

Figure 6–21: A diagonal is confined to the length of the shorter axis.....6–103

Figure 6–22: Mapping produced by `1 2 2` *transpose of a rank-3 array*.....6–104

Figure 6–23: Mapping produced by `1 1 2` *transpose of a rank-3 array*.....6–104

Figure 6–24: Mapping produced by `2 1 2` *transpose of a rank-3 array*.....6–105

Figure 6–25: Take and drop.....6–108

Figure 6–26: Effects of alternative left arguments to grade.....6–114

Figure 6–27: Effective weight of characters occurring twice in ⍺.....6–117

Figure 6–28 : Expression to select an item from a rank-3 array. 6–127

Figure 6–29 : Expression to select a sub-array from a rank-3 array. 6–128

Figure 6–30 : Indexing an axis by an item removes that axis from the result. 6–129

Figure 7–1 : Verbs derived from the slash adverbs. 7–7

Figure 7–2 : Identity elements for scalar dyads. 7–9

Figure 7–3 : Conjunctions. 7–15

Figure 7–4 : Composition templates. 7–16

Figure 7–5 : Three forms of composition. 7–17

Figure 7–6 : Short forms of rank specification in ⍤. 7–19

Figure 7–7 : Arguments of the cut conjunction and of its derived verb. 7–22

Figure 7–8 : Verbs derived from the dot conjunction. 7–27

Figure 8–1 : Comparison of ∇*-editor and* `⎕fx` *or* `⎕fd`. 8–2

Figure 8–2 : A typical line edit using the ∇*-editor.* 8–9

Figure 9–1 : The visible referent of a name is the first appearing in `)sinl` 9–6

Figure 10–1 : Effective value of `⎕ec` *is the maximum* `⎕ec` 10–9

Figure 11–1 : Shared variable interactions between tasks. 11–1

Figure 11–2 : Possible values of `⎕svs` 11–10

Figure 11–3 : Transition diagram for state changes. 11–15

Figure 12–1 : Schematic representation of T- and S-tasks. 12–1

Figure 14–1 : Table of file permission codes. 14–12

Figure 15–1 : Internal representation of APL characters. 15–9

Figure 16–1 : System nouns, indicating which you can set. 16–1

Figure 16–2 : Tolerant and intolerant comparisons. 16–11

Figure 16–3 : Behavior of errors depending on `⎕ec` *setting.* 16–14

Figure 16–4 : Event numbers. 16–16

Figure 16–5 : Summary of `⎕fmt` *format phrases.* 16–29

Figure 16–6 : Summary of `⎕fmt` *qualifiers.* 16–32

Figure 16–7 : Summary of `⎕fmt` *decorators.* 16–33

Figure 16–8 : Text-delimiting characters for decorator text. 16–33

Figure 16–9 : Possible values of a `⎕svs` *result cell.* 16–77

Figure 17–1 : List of System Commands. 17–1

Figure 17–2 : APL characters on an ASCII terminal with `)cc normal` 17–6

Figure 17–3 : Conditions in a clear workspace. 17–6

About this Manual

This manual describes Version 20 of the SHARP APL system and language. It is written for people who wish to use the SHARP APL system or who wish to read or write programs in the APL language provided by the system. It assumes no previous programming experience in APL or any other language. The manual provides elementary discussions to introduce each topic and gives a complete description of each of the features of the language and the system. Both the novice and the experienced programmer should find it a useful reference. This edition of the SHARP APL Reference Manual supersedes all previous editions. It includes descriptions of all enhancements and extensions introduced since the last publication of the manual (1981). It also includes brief descriptions of all SHARP APL modules that are packaged and documented separately; for example, auxiliary processors and session handlers.

Related Publications

"A Dictionary of APL," by Dr. K.E. Iverson, is a useful companion to this manual. The dictionary, published in APL Quote Quad, vol. 18, no. 1, provides a concise description and summary of Iverson's proposals for APL. It includes some discussion of "dialects," that is, different APL systems. It is not specific to SHARP APL, and includes some features that are not currently provided in SHARP APL. You may also find it helpful to consult some of the other SHARP APL publications that are available. A complete list of these publications can be obtained through your SHARP APL customer representative.

Conventions Used in this Manual

Unless otherwise noted, examples in this manual are based on an index origin of 1 (`⎕io←1`). Examples of input to, and output from, SHARP APL use the first and second alphabets of the APL character set defined in `⎕av`. The alphabets used in this manual may not look the same as those that appear at your terminal or printer. To determine what these alphabets look like at your terminal or printer, display the following locations in `⎕av`:

```
      ⎕av[86+⍳26]
abcdefghijklmnopqrstuvwxyz
      ⎕av[113+⍳26]
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

What You Should Not Expect from this Manual

To use a program that manages and interprets for itself its interactions with the user – that is, presents its own *user interface* – you do not need an APL manual but a manual for that application. The present manual is relevant when you the APL system without the benefit of an all-embracing program, or when you write new programs or modify existing ones. This manual is for those who will see and use the underlying APL as well as the applications built on it.

1 *Overview*

SHARP APL is a high-performance interactive terminal system for the APL language. It runs on IBM S/370-compatible mainframe computers, under IBM's MVS operating system, and on IBM-compatible personal computers running DOS. On a mainframe, it can be used from any of a variety of terminals or from a microcomputer emulating the behavior of a terminal. The terminal or emulator may be directly connected to the mainframe, or linked to it by a data or telephone network or through a gateway from a local-area network. SHARP APL may also be accessed from batch programs, such as BAPL.

In general, each of the successive versions of SHARP APL has been compatible with its predecessors. With only a few minor exceptions (noted later in this chapter), a program written for an earlier version of SHARP APL executes in the current version without change.

APL: Visible and Invisible

The next part of this chapter shows what you would see while executing a few simple sentences in APL. It shows what you would see at the screen or paper while seated at a keyboard, interacting directly with the SHARP APL interpreter, writing instructions in the APL language.

That is not the only way, or even the common way, to use the SHARP APL system. In many situations the APL language is not apparent. From start to finish, the entire session may be managed by programs that control the screen and direct the work without asking for a single statement in the APL language or displaying a single APL character. An application that is in fact written in APL may simulate some other language; usually, the application provides a language of its own, specific to the task at hand. You could remain unaware that the program you are using is written in APL and is being interpreted by an APL system.

There are also batch APL programs that run alone, perhaps in the middle of the night, carrying out large jobs automatically, on schedule, with no terminal connected and no human input at all. They may be interrupted by scheduled shutdown or by system failures, but after interruption can be automatically restarted so that work is resumed without loss.[1]

[1] The principles behind batch APL tasks on the SHARP APL system are briefly described in this manual. For details, see the separate publication, *Batch Tasks in SHARP APL*.

Introduction by Example

Looking over the shoulder of someone using SHARP APL might reveal the following. Although brief, it illustrates several points about the APL language and the way APL can be used.

```
      2×3
6
      5280÷16.5
320
      prices←6 4 2.5 3 ⊣ orders←3 1.5 0 2
      ⊢costs←orders×prices
18 6 0 6
      +/costs
30
      )copy 19 ddef
saved 1984-11-26 20:15:30

      def ⋄ def ⋄ def
sum: +⌿ω
N: ''⍴1↑⍴ω
average: (sum ω)÷(N ω)

      sum costs
30
      'Costs:' ⊃ costs ⊃ '   Total:' ⊃ sum costs
Costs: 18 6 0 6    Total: 30

      average costs
7.5

      trans←⎕av[256?256]
      ⊢encoded←trans[⎕av⍳'Hello. This is a message.'
mCDD{$∇⍞⍞+⎕∇+⎕∇ω∇gC⎕⎕ω⍞C$

      ⎕av[trans⍳encoded]
Hello. This is a message.
```

Figure 1–1 : Elementary examples to illustrate some aspects of APL.

- ***Interaction.*** At the terminal, the lines of this example appear one after the other. You enter the first from the keyboard:

```
2×3
```

  As soon as you press the *Enter* key,[2] SHARP APL executes the sentence you have just entered. The result `6` appears, and the cursor moves down.

- ***Dialogue.*** To show that it is ready for new input, SHARP APL moves the cursor from the left edge to six positions to the right. Now you can enter something else. The dialogue is clear because what you type is indented and what the computer replies is not.

- ***Parts of speech.*** The grammar of APL is described in terms like those used to describe natural language. In the example, you can see the principal parts of speech:

  - *Nouns* are numbers (such as `2` or `3 1 0 2`) or characters (such as `'abc'`. To indicate that what you write is a character, or a set of characters, you place a quote mark (`'`) at the beginning and another at the end, as you do in English.

  - *Pronouns* are names arbitrarily assigned to nouns. In the example, `orders` is a pronoun that stands for the noun `3 1.5 0 2`, and `costs` is a pronoun assigned to the result of multiplying `orders` by `prices`. A pronoun is thus equivalent to what mathematicians call a *variable.* It is variable in the sense that (like a pronoun in natural language) it does not have a fixed meaning, but can be associated with different nouns at different times.

  - *Verbs* (such as ÷ or ×) act upon nouns or pronouns.

  - *Adverbs* (such as /) and *conjunctions* (such as ∘.) apply to verbs to produce related verbs. For example, in +/ the adverb / modifies + (addition) to produce a verb you might call "summation." The dot conjunction . produces a verb which performs inner product on matrices or tables.

  - *Is* (or, as grammarians say, the *copula*) links a thing and some attribute. APL uses ← to link a pronoun and the name (or pronoun) assigned to it. A sentence such as

```
rate←1.6
```

    may be read as "`rate` is 1.6."

[2] Depending on the device, it may be marked *Enter* or *Return* or perhaps *Send* or ↵. This manual uses *Enter* to mean any of those, as appropriate to the device.

- ***Collective nouns.*** A noun may be a single number or character or a collection of many numbers or many characters known by a single name. Because the members of such a collective noun are arranged rectangularly, a noun is said to be an *array*.[3] Collective nouns include *lists* or *tables*, as well as higher-rank arrays.[4] The simplest possible array has no rows or columns, and consists of a single item. APL allows direct use of a collection. That is, in many situations, an APL sentence is written the same way regardless of whether a noun in the sentence consists of a single item or many items.

- ***No declarations.*** Some programming languages require that you state in advance the type and shape of data to which a name will refer. In general, SHARP APL does not use declarations.[5]

- ***Display.*** SHARP APL automatically displays the result of a calculation (unless you tell it do something else with the result). It is not necessary to request display or to tell SHARP APL how to format it (although you may, when you want a different format).

- ***User-defined verbs.*** New verbs can be created and given arbitrary names (such as `sum` or `def` in the example). In other respects, they are used in the same way as other verbs.

- ***Community resources.*** You can build on work done earlier (your own, or the collected work of others) by making use of the SHARP APL system's *libraries* of saved work. The example shows use of the command `)copy` to bring in a collection of user-defined verbs that includes `def`.[6]

- ***Programs.*** Programming is the creation of new verbs. In other languages, these may be called programs or functions or procedures. In the example, a verb `def` is used to create the new verb `sum` from the characters `sum: +/⍵`. The SHARP APL system also provides an editor to help you enter or revise definitions.

- ***Nested definitions.*** A verb may refer to other user-defined verbs in its definition. For example, the definition of `average` makes use of the verbs `sum` and `N`. Verbs can refer to each other almost indefinitely. A verb can also refer to itself; a verb that does so is said to be *recursive*.

---

3 The word *array* originally referred to a drill formation of soldiers, marching in rows and columns.

4 In mathematical parlance, a list is often called a *vector*, and a table is called a *matrix*. While previous editions of the SHARP APL Reference Manual used the mathematical terms, this edition instead uses the more informal terms.

5 There remain some traces of the notion of declaration. For example, a user-defined verb may declare a list of names to be *local* to its definition. But SHARP APL makes much less use of declaration than such languages as BASIC or C, and has no requirement to state in advance the type or shape of an array.

6 The example does not show how `def` was defined; for the listing of a program that accepts definitions in the form shown, see the Δ program defined in Table 3 of "A Dictionary of APL," page 39.

- ***Reading and speaking.*** Because of the similarities between the grammar of APL and the grammar of a natural language such as English, a well-written APL sentence may be read aloud with clarity. For example:

```
      prices←6 4 2 3                    "Prices are 6, 4, 2, and 3."

      -/(⌈/prices),⌊/prices             "Difference between the
4                                        maximum and minimum
                                         prices."

      +\odd←¯1+even←2×+\6⍴1              "The cumulative sums
1 4 9 16 25 36                           of odd, which is 1 less
                                         than even, which is
                                         twice the cumulative
                                         sums of 6 ones."
```

Arrays

With the exception of packages, every noun is an *array*. SHARP APL also provides a noun called a *package*.[7] An array is composed of any number of *items*, arranged along any number of *axes* or *dimensions*.

There is no special procedure for saying "This is an array," since every noun is automatically an array. An array that contains only one item and has no axes at all is just a particular case of an array, called a *scalar*.

When a verb refers to a noun, it refers to the array *as a whole*. Depending on what the verb is and how it is used (and how it may be modified by adverbs), the verb either treats the entire array collectively, or considers the array to be a set of cells, so that the verb applies to each of the cells in *parallel fashion*.

Arrays are *homogeneous*. That is, all the items within an array must be of the same type. The type of item within an array determines the type of array.

An item – and thus an array – may be:

- *Numeric*. Numbers in SHARP APL may be Boolean, integer, real, or complex.
- *Character*. For example, `'x'`, `'cost'`.
- *Box*. A box is a single item that contains an array *enclosed* within it. As long as the box remains closed, it can occupy a single position in an array, just as a single number or a single character can. Boxes

[7] A *package* is a special type of noun with some useful properties for treating disparate objects together. A package is not an array – its various members are identified by name rather than by position. Packages are described in Chapter 4, "Nouns and Pronouns".

permit data of differing types and irregularly shaped data to be stored in a regular rectangular framework.

Origins of APL

APL grew from proposals for a notation for the representation of algorithms put forward by Dr. Kenneth E. Iverson and published in his seminal work *A Programming Language*.[8] In 1966 the initial letters of the book's title were adopted as a name for the language.

Dr. Iverson's proposals extended existing conventions of arithmetic and algebra to embrace computation. He made extensive use of common mathematical symbols. In addition, he followed standard mathematical practice in assigning meanings to existing special symbols (for example, the Greek letters iota, rho, and epsilon) and in coining new but mnemonic symbols (for example, ⍉ for transpose).

Iverson's notation existed – and continues to exist – independently of computer systems to execute it. It provides a ready means of expression wherever people need to discuss formulas or procedures: in books, on blackboards, or even the backs of envelopes. Publications written in APL[9] appeared before any attempt had been made to build an APL computer system. APL is used in several texts simply as a vehicle for describing a procedure to human readers. The attraction of the language is that it provides a rich set of convenient and powerful operations as primitives, each denoted by a single symbol. Because of that, APL permits very concise statement of common procedures from business, finance, engineering, mathematics, and so on.

Continuing Divergent and Convergent Development

Although the core of the APL language goes back to proposals made many years ago, the language continues to grow and develop. Contributions come from the proprietary (and hence different) extensions of various builders of APL systems, both commercial and academic. The language also benefits from the unification and standardization work of the international APL community. Several years of discussion and refinement resulted in the 1987 publication of the ISO standard for the APL language.[10] SHARP APL is consistent with the ISO standard. SHARP APL also contains a number of services and extensions not covered by the standard.

8 New York: Wiley, 1962.

9 For example, Iverson, Sussenguth and Falkoff, "A Formal Description of System/360," *IBM Systems Journal*, vol. 3, November 1964.

10 *Standard for Programming Language APL*, International Standards Organization, ISO/IEC/IS8485.

In 1987, 25 years after the seminal book, Iverson published "A Dictionary of APL."[11] This publication concisely presents an overall view of a number of important APL innovations, some but not all of which have been included in SHARP APL. Wherever appropriate, this manual adopts the terminology used in "A Dictionary of APL." It also makes brief mention of some of the proposals in the "A Dictionary of APL" that are not currently provided in SHARP APL.

Development Related to Array Processing

APL borrowed from matrix algebra the idea that an operation may be defined on an array as a whole rather than on each of its elements separately. Because of this, many procedures that in other languages require indexes and loops can be written much more simply in APL. Several texts take advantage of this fact to present parallel discussion in matrix algebra (for continuity with the published literature, for example in economics or statistics) and in APL (for immediate execution at an APL device).[12]

Since so much of data processing involves the analysis of data arrays, the fact that APL operations apply directly to arrays often makes APL programs simpler and shorter than programs written in other languages. Because of its array orientation, APL requires much less iteration, looping, testing, and branching. APL lacks constructs that correspond to the `DO`, `FOR`, or `WHILE` instructions of some other languages.

Also because of its array orientation, APL is rich in verbs for the manipulation of arrays: shape, reshape, rotate, transpose, take or drop part of an array, compress, expand, replicate, and so on. Because these array operations are little described in conventional arithmetic, there are no established symbols for them; verbs dealing with array manipulation account for many of APL's coined symbols.

APL includes provision for *indexing*; that is, selecting individual items from within an array. However, APL programs often make less use of indexing than would be required in languages whose verbs are defined only on single items. Moreover, the APL interpreter is designed for efficient processing of verbs applied to an entire array, so that a program written to exploit the extensive array-handling powers of APL usually runs faster than a program defined iteratively for individual elements.

A number of recent additions to the language continue to develop the treatment of arrays. Foremost of these is the concept of *rank*, and the related ideas of *cell* and *frame*. Briefly, an array of data is considered to consist of *cells*. Each cell provides the data for a single case. For example, a

11 Association for Computing Machinery, *APL Quote Quad*, vol. 18, no. 1, 1987.

12 For example, *APL\360 with Statistical Applications*, by Keith W. Smillie, Addison-Wesley, 1974.

single case might be a table, requiring two axes (rows and columns). The cells are arranged in a *frame*, which may itself have any number of axes. An operation is applied throughout the frame. It is applied automatically to each of the cells without the need to write loops, tests, or iterations. The concept of rank is pervasive in "A Dictionary of APL," and widely mentioned in this manual,[13] even though SHARP APL does not provide all of the uses of rank described in "A Dictionary of APL."

Another development in the language is the addition of *boxes*. A box is an item. It occupies a single point in an array, just as a number or a character occupies a single point. However, when you open the box, it may contain an array of any type or shape or size. Thus, the regular structure of an array may contain other arrays within it.[14]

Interpretive Basis for APL Systems

APL systems – SHARP APL included – are usually interpretive rather than compiled. That is, a program written in APL is stored in the machine in more or less the form in which it was written. The machine does not attempt to attach meaning to the words or symbols, or to assign space in which to store data, until a sentence is actually executed.

There are some costs in delaying interpretation until the moment of execution. There are also important benefits. The costs are minimized when programs make use of APL's power to deal with arrays, since a single symbol or expression may invoke a great deal of processing.

The great advantage of interpretation to the developer of a program, or to an informed user, is that it is so easy to make corrections or changes to the program as it goes along. Each time the program halts, whether in response to an error or to an interrupt signaled from the keyboard, you can display the values of intermediate results, look at or modify the program, and then resume work. It is even possible (although not necessarily a good idea) to start running a program before many of its parts are defined and fill them in as the need arises.

13 Examples of the uses of rank are given in R. Bernecky, "An Introduction to Function Rank," , Association for Computing Machinery, *APL Quote Quad*, vol. 18, no. 2, 1988.

14 In the APL literature, an array whose members are boxes has been variously called a *general* array, a ***nested*** array, an *enclosed* array, and an *array of arrays*.

Access to Files

A *file* is a collection of data that is outside a computer's active working memory (usually in some more permanent storage) but accessible to it. A file may be of any size, possibly much larger than the available working memory. Because of its unpredictable size, the task of transferring information between a file and working memory is usually split up and handled a segment at a time.

SHARP APL provides access both to specialized files created expressly for use with APL programs and to standard files used by other programming systems under the general umbrella of the MVS operating system.

Shared Work and Private Work

As a user of the SHARP APL system, you get a private space in the computer (your *active workspace*) and a private collection of stored workspaces and files (your *library*). You are insulated from other users, even though there may be many of them. The SHARP APL system makes it appear that you are the sole user of an APL machine.

You may also interact with other users. Your task may communicate with the tasks of other users or with other APL tasks of yours that are running at the same time. You can arrange with other users to share access to some files, even making concurrent revisions to them.

Summary of Features of SHARP APL

Features Beyond Those Required by the ISO Standard

Shared variables A noun is shared when it is visible (and can therefore be set or used) by two otherwise independent tasks. The ISO standard does not require shared nouns, [15] but prescribes the mechanism to be used if an APL system includes them. Although the feature should properly be called *shared nouns*, the term *shared variable* is in more common use, and will be used in this document. SHARP APL includes complete support for shared variables, plus the extension `□SC`, which permits one task to wait until the other does something or until a time limit runs out.

15 This feature is called *shared variables* in the standard.

APL component files SHARP APL permits the creation of files that are divided into *components*. Each component is an APL noun (of any type, size, or shape). Data is transferred between the file and the workspace of an APL task one component at a time. A component is identified by its index number in the file. Files are accessed by *file verbs*, built in as part of the SHARP APL language. The APL component file system is described in Chapter 14, "APL Component Files"; the various file verbs are included in Chapter 16, "System Nouns and Verbs".

Formatting The format verb ⍕ is extended to include "format by example." The system verb `⎕fmt` formats arrays according to control statements akin to those used in Fortran or Cobol.

Batch APL tasks Any APL task may start an APL task which then runs without further direct control. The new task may be started either directly by the parent task or by filing a request with the *Batch Task Scheduler*, which may run the task at a time when the requesting task is no longer active.

Shared variable tasks Any APL task may start another task, controlling the new task by way of a name shared with the APL S-task processor rather than from the keyboard.

Constrained and profiled users The system administrators can designate individual user numbers *constrained.* This is usually done to simplify work for people who only use applications prepared by others and never write sentences directly in APL.

If you are a constrained user, as soon as you sign on, the SHARP APL system automatically loads your `Profile` workspace and starts executing the latent expression stored there. [16] As soon as the application ends (or is interrupted by an error), the system signs you off.

If you are a profiled user, as soon as you sign on, the SHARP APL system automatically loads

16 See Chapter 16, "System Nouns and Verbs" for the definition of the latent expression, `⎕lx`.

your `Profile` workspace, unless your previous session terminated abnormally, in which case it loads your `continue` workspace. It then starts executing the latent expression found in that workspace. When the application ends (or is interrupted by an error, control returns to you.

Crash recovery The SHARP APL system routinely copies all work in progress to auxiliary storage every few seconds, so that in the event of a system crash work can be restored with at most a few seconds' loss.

Disconnect recovery When a keyboard task is interrupted by a communications failure, its work is automatically saved.

Arbitrary input/output SHARP APL provides verbs that directly control the characters transmitted to a remote device.

Boxes New since the last edition of the SHARP APL Reference Manual; see below.

Complex numbers New since the last edition of the SHARP APL Reference Manual; see below.

Event handling New since the last edition of the SHARP APL Reference Manual; see below.

Packages A type of noun called a *package* contains objects each of which is identified by name (rather than by its position within an array structure). The objects may include both nouns and verbs, making packages particularly useful for program overlays. The SHARP APL language includes verbs for forming, manipulating, and extracting objects from packages.

Composition, rank, and cut conjunctions New since the last edition of the SHARP APL Reference Manual; see below.

Determinant New since the last edition of the SHARP APL Reference Manual; see below.

Features Added Since the Last Edition

Formatting verbs The *thorn* verb ⍤ is extended to take a character left argument, permitting "format by example." The definition is consistent with that adopted by

IBM in APL2. Thorn also accepts an empty list as its left argument, which provides a dyadic form of expression for monadic thorn. This is useful in formatting expressions using the *under* conjunction.

The system noun `⎕ps` affects the position and spacing of an array whose items are boxes.

Boxes and boxing verbs An array of data may be *boxed*, so that it is treated as a single item within another array. Verbs are provided for enclosing an array to form a box and for opening a box. There are related extensions for formatting the display of boxed data.

Complex numbers Complex numbers are supported as a type of numeric data and may be displayed or entered from the keyboard. The letter `j` separates the real and imaginary parts. Arithmetic operations on real numbers return complex results where appropriate; some expressions that formerly produced a `domain error` no longer do so. The domains of all appropriate verbs are extended to include complex numbers.

Event handling An *event trap* is a statement that states in advance the action to be taken when various events (errors or interrupts) occur. In addition to those events automatically reported by the interpreter, a program may also signal an event to be trapped. Event handling is described in Chapter 10, "Event Handling"; the system names related to event handling are included in Chapter 16, "System Nouns and Verbs".

Composition, rank, and cut conjunctions The conjunctions ⍤ ⍥ and ⍠ are introduced. They form *compositions* which apply a pair of verbs jointly to each cell of a data array. Applied to a verb and a noun, the *rank* conjunction determines how a noun's axes are divided into cell and frame axes. The *cut* conjunction partitions (or "cuts") an array into cells to which the verb applies. Conjunctions are described in Chapter 7, "Adverbs and Conjunctions".

Replicate adverb The adverbs / and ⌿ are extended so that they not only select cells (when the left argument is made up of 1's and 0's) but may replicate cells (where the left argument calls for more than one).

Generalized determinant The algebraic functions *determinant* and *permanent* are provided as derived verbs by using −.× and +.× monadically. Using other verbs with the dot provides a form of generalized determinant.

File hold by range The file verb ⎕*fhold* is extended so that cooperating tasks may coordinate their access to specific ranges of components within a shared file. In addition, additional files may be held or unheld without affecting other previously held files.

File creation The file verb ⎕*create* now allows specification of a starting component number for a file.

Function monitor The system verb ⎕*fm* permits precise recording of the CPU time and elapsed time of user-defined verbs, line-by-line or overall. It also provides a count of the number of times each line of each verb was executed.

Symbol table resized The symbol table (the workspace's directory of all names in use in it) is automatically resized when it needs more space. Formerly, this required manual intervention.

New password at sign-on You can now change your password at the time you sign on. It was formerly possible only at sign-off.

Extended take and drop The *take* (↑) and *drop* (↓) verbs have been enhanced to permit a short left argument. For example, 6↑2 3 4⍴5 will provide a result of shape 6 3 4.

First axis catenate The dyadic form of the *join* verb (⍪) provides catenation along the first axis.

Table verb The *table* verb (⍪) creates a rank-2 table from any array. In particular, it forms a one-column table from a scalar or list argument.

Array and string search The *in* verb (∈) finds all occurrences of one array within another array. It is an excellent way to perform string searching in text.

Nubsieve The *nubsieve* verb (≠) can be used to determine the unique major cells within an array.

Continue workspace name The `⎕twsid` system verb provides the capability to specify the name of the workspace saved by the system when your session terminates abnormally.

File library access controls The file system verbs `⎕rdacl` and `⎕stacl` allow you to associate an access matrix with your entire file library, as well as with individual files. A new capability provided by these enhancements permits you to allow another user to create files in your library.

Enhanced disclose The *disclose* verb (>) now provides padding of the result, so that arrays of differing rank and shape may be disclosed without error.

Raze The *raze* verb (↓) provides a capability similar to disclose, except the disclosed arrays are assembled along the leading axis rather than being laminated.

Shutdown notification To ensure that applications are able to terminate in an orderly manner when the SHARP APL system is being shut down, tasks are now notified prior to shutdown. The result of the workspace reporting verb `2 ⎕ws 3` is changed to indicate that APL shutdown is in progress. In addition, SHARP APL now issues a `⎕sc` post to all tasks at that time.

Effective workspace size `2 ⎕ws 3`, a workspace reporting verb, now contains information as to effective workspace size. This information is useful for applications that use a workspace paging system such as LOGOS.

Hot continue notification `2 ⎕ws 3`, a workspace reporting verb, now contains information as to whether the previous session saved a *hot continue* workspace.

SVP Extensions The shared variable verbs `⎕svn`, `⎕svo`, and `⎕svq` have been enhanced to provide support

for NSVP, the Network Shared Variable Processor.

User-defined verb definition The *fix* verbs (`□fx` and 3 `□fd`) now permit user-defined verbs to contain blank lines. This simplifies the porting of APL applications between APL systems. In addition, suspended and pendent verbs may now be fixed.

System editor The *system editor* for user-defined verbs has been enhanced to reduce the occurrence of `si damage`.

Features Changed or Discontinued

AMPX interface AMPX, the terminal interface that provided access to SHARP APL systems for asynchronous terminals, has been replaced by the ATH/E terminal handler.

)terminal The DCTTAPE and IBM 2741 terminals are no longer supported by SHARP APL.

Arbitrary input/output of tables The verbs `□arbout` and `□arbin` are now restricted to lists or items. (They formerly accepted table arguments.)

No idles for fast terminals Terminals running at 1200 baud or faster are assumed to require no idles. Idles are transmitted for the benefit of mechanical terminals without buffers, which require time to execute tabs, carriage returns, etc. Idles are still transmitted for terminals running at less than 1200 baud, unless suppressed by the `)terminal` command.

IBM 2741 Support Discontinued Terminals using the IBM 2741 communications protocol are no longer supported by SHARP APL.

2 *Tasks, Users, Libraries*

SHARP APL executes instructions written in the APL language. The instructions may be typed a sentence at a time from a keyboard for *immediate execution* or contained within a *program* (which, in APL, is also called a *user-defined verb*).

Timesharing

SHARP APL does its work within the context of a *timesharing system.* At a central computer (called the *host*) a program called a *supervisor* presides over work that is being done concurrently for many different people. Work on the various jobs is in fact done serially (since each central computer is fundamentally a serial device). The computer handles a little of each person's job in turn, taking them in rotation. The supervisor allocates a small amount of time to one job, then sets that aside and switches to another. It is able to do this so rapidly that each job appears to run smoothly and continuously.

To permit many people to work at the same time, the central computer has one or more devices called *communications processors.* A communications processor permits the timesharing supervisor to be connected to the various people who are using the machine. The connection may require little more than a meter of wire, or it may involve many interconnections, relayed through a network spanning thousands of kilometers.

At the other end of each such link, the person controlling the job works from a device called a *terminal.* The terminal usually has a *keyboard,* at which the user can type sentences to be sent to the computer, and a video screen or paper printer, at which the computer can display its replies.

Users

A person enrolled in the SHARP APL timesharing system and thus entitled to use it is said to be a *user.* The system identifies each user by a number called a *user number.* In the SHARP APL system, a user number usually has seven digits; but it may have as many as ten digits or as few as four.

The system maintains a file in which it keeps a record of the name associated with each user number, various usage statistics, and resource *quotas.* A

quota limits the user's access to certain facilities. These limits are discussed further in the chapters that deal with those facilities.

Users: Ordinary, Privileged, Constrained, and Profiled

Certain levels of authorization are tied to the user number. In general, this manual describes the facilities available to the *ordinary user*. There are three other levels of use.

Certain members of the system's administrative staff are *privileged users*. A privileged user is exempt from a few of the restrictions that otherwise apply to all users. For example, a privileged user can modify a workspace in a public library or can save a workspace into the library of another user. By using these privileges upon special request, the system administrators are sometimes able to correct situations that an ordinary user could not remedy.

Some users are designated *constrained users*. A constrained user is able to run application programs prepared by others, but never sees or uses the APL language directly. Enrolling as a constrained user is appropriate when:

- the application you want to use is set up for constrained use (and never requires you to use APL directly)
- you do not want to see the APL definitions of the programs you use.

Being a constrained user of SHARP APL is much like being an ordinary user of most other timesharing services. That is, you can use applications prepared for you by others, but you cannot modify them or write your own.

For each constrained user, the system administrators establish a *profile*. The system automatically activates the profile each time a constrained user signs on. The profile is contained in a workspace, for instance the user's workspace called `Profile`.[1] The system automatically loads the profile workspace when the task is initiated. Doing so starts an application immediately or offers a menu of alternative applications for the user to choose from. When whatever the profile workspace does comes to an end (either because you conclude it in the normal manner or because something goes wrong for which the application makes no provision), the session ends.

A *profiled* user is a user for whom the system automatically loads a profile workspace at sign-on but is in other respects an ordinary user.

Only the system administrators can set or change a user's level of authorization.

[1] The administrators may adopt some other standard name for the profile workspace.

Tasks

A session of work that the central computer carries out for a user is called a *task.*[2] A user may have several tasks running at the same time. Usually only one of them is directly linked to a terminal. Once a task has been started, it may itself start other tasks.

The SHARP APL interpreter deals with each active task for a brief period (a fraction of a second) and then turns as needed to other tasks, thus sharing its time among them. Some tasks have higher priority than others. That is, when there is contention for the central processor's time, a low-priority task may have to wait longer for its next burst of activity and thus may appear to run less quickly than a task that has higher priority. The system gives highest priority to interactive tasks (those directly controlled by a human user or another task).

In general, tasks are independent. Tasks owned by different users are completely invisible to each other. There is only one situation in which a task has direct knowledge of what another task is doing and is directly able to control that task: when the first task has initiated the second one. (See S-tasks, below.)

Every task has its own working environment, called its *workspace* or *active workspace.* Normally, events in one task's workspace cannot affect events in another task's workspace. However, a pair of tasks may arrange by mutual agreement to share a name between them. When they both do that, a value that one of them sets in its own workspace is thereby set in the other task's workspace. In this manner a pair of tasks may communicate. (See Chapter 11, "Shared Variables".) Several tasks may also agree to make use of the same file, either at separate times or concurrently, and thereby exchange information.

Each task has an *owner.* The owner is the user number supplied by the person who signed on to start the task or the user number of the task that started it or is responsible for it. The owner's user number is part of the name of every APL workspace or APL file.

Types of Tasks

As a user of the SHARP APL system, you may have any of the following four types of tasks:[3]

T-task *Interactive* task controlled by a human user from a terminal.

---

2 Probably the computer also does work unrelated to APL. To be more explicit, the system administrators may speak of an "APL task" in contrast to some other kind of task; in this manual, the only tasks discussed are those that involve APL, so we refer to any of them just as a "task."

3 A fifth type, R-task, is used for communication with a special device, such as a tape drive or a printer, and is outside the scope of this manual.

S-task *Interactive* task, but without a terminal, controlled instead by a name shared with another task.

N-task *Non-interative* task, without a terminal, for *immediate* execution, or for *scheduled* execution at a precisely specified later time.

B-task *Batch* task, without a terminal, for *delayed* execution at the convenience of the system's Batch Task Scheduler, within constraints that the requester can set.

Every Task Has Its Own Active Workspace

When a task starts, SHARP APL allocates a block of the system's storage to serve as the task's *active workspace*. Depending on the task's level of activity, the active workspace may reside in the system's main storage, or it may reside on auxiliary storage, such as disk. The active workspace is the place in which all the task's computation takes place. Any program or data involved in a calculation must be present in the workspace at the moment the calculation is performed.

The workspace serves both as the *working storage* in which calculation occurs and the *environment* in which names are understood. In general, a user-defined verb can refer to any verb or noun whose definition is present in the same workspace. However, during execution verbs may establish their own *local names*, which are hidden from general view.

Keeping nouns and verbs together in a workspace makes it easy for them to refer to each other. You can combine several verbs in a single sentence. The definitions of one user-defined verb can refer to another user-defined verb or to user-created nouns. The fact that they are all stored in the same workspace makes it unnecessary to *link* the definition of a verb with the names that it uses (although in some other languages that is required when separate programs work together).

Saved Workspaces

The SHARP APL system permits you to set aside a copy of everything in the active workspace. Such a snapshot of the active workspace is called a *saved workspace*. Development of an APL application usually involves bringing together the various definitions it will use and then saving them together as a saved workspace.

The command `)save` records the contents of the active workspace to permanent (but inactive) storage. Also, at the end of a non-interactive or batch

task or when an interactive task is interrupted by a failure of communications or of the system itself, the SHARP APL system automatically saves the task's active workspace.

Size of the Active Workspace

Ordinarily, all workspaces are the same size. (Having them all the same size makes it easier for the timesharing system to schedule the "swapping" of workspaces into and out of its active memory.) The standard size of an active workspace is established by the system administrators.[4]

However, the system administrators may create a workspace of a non-standard size. Usually this is done to provide additional space for an application that develops particularly large temporary arrays. When such a workspace is saved, its additional size is saved with it. No special request or permission is required to load such a workspace. Because a large workspace requires more resources from the computer system, response may be somewhat slower or more uneven when such a workspace is used.

The life of a task is identical to the life of its active workspace. That is, every task has exactly one active workspace, and the active workspace remains in existence as long as the task continues, and no longer.

Ways to Start a Task

A task can be started in a number of different ways:

- By signing on from a terminal. That starts a T-task.
- By signing on from an already-running task, through a name shared with AP1. That starts an S-task.
- By executing `□run` from an already-running task. That starts an immediate N-task.
- By submitting a request to the Batch Task Scheduler from an already-running task. That will start a B-task or a delayed N-task at some later point in time.

Duration of a Task

Once running, any task may be forced to terminate by the verb `□bounce` executed by the system administrators or by a task having the same owner.

An interactive task (T-task or S-task) starts when it is *signed on.* The sign-on transmission includes the owner's user number and password. The task

[4] The storage consumed by the active workspace is reported by the system verb `2 □ws 4`; the number of bytes remaining unused in the active workspace is reported by `□wa`. See Chapter 16, "System Nouns and Verbs".

runs until the person or task that started it signs it off with the command `)off` or `)continue` (see Chapter 17, "System Commands") or until it is forced off (see below).

A batch task (N-task or B-task) may either start immediately, when its parent task executes the verb `□run` to start it, or at some later time, when the Batch Task Scheduler starts it in accordance with a request that the parent task submitted earlier. Once started, a batch task runs until it completes its initial instruction, or until it encounters an untrapped error, solicits input, or is forced off.

The Batch Task Scheduler has the power to restart a B-task or N-task that has been interrupted by a system shutdown or crash. When you submit a request for a B-task, you indicate whether the Scheduler should restart it following a normal termination (such as shutdown) or an abnormal termination (system crash).

When you indicate that it may restart after a normal shutdown, you give the Scheduler greater latitude in scheduling your task. When you indicate that your task can be restarted following a system crash (a rather unlikely event), you are asserting a higher degree of confidence that your programs can be correctly resumed following an unanticipated interruption. Batch tasks are described in Chapter 13, "Batch Tasks", and the system verb `□run` in Chapter 16, "System Nouns and Verbs".

Work Saved at Forced Termination, Line Drop, or Crash

The SHARP APL system automatically saves the active workspace of a task that has been bounced.[5] The termination workspace from a bounced N-task or B-task is given the name specified when the task was started. The saved workspace from an interactive task that has been bounced or has terminated prematurely by a line drop is normally given the name `continue`.[6] The system verb `□twsid`, described in Chapter 16, "System Nouns and Verbs", provides control over both the name of the termination workspace and whether or not the system will save it at task termination.

A similar mechanism protects you from the risk that the contents of your active workspace might be lost in the unlikely event that a task is cut short by a system failure (or "crash"). If this does happen, as part of restarting SHARP APL, the system recovers and saves the active workspace of each task, naming it in the same way as a workspace saved following a bounce

5 "Bounce" – meaning to force the termination of a task – is another of those informal phrases for which no one seems to have a polite or official synonym.

6 Watch out: If you have multiple interactive tasks (for example, a T-task that is running an S-task), unplanned termination of more than one task causes more than one request to save `continue`. Later versions of `continue` overwrite earlier ones. The exact order in which the system deals with them, and hence the order in which `continue` workspaces are saved, is unpredictable.

or line drop. With this safety mechanism, the most you can lose is what you did in the last several seconds.

The next time you start an interactive task, SHARP APL automatically loads the `continue` workspace. The effect is that you resume work at the point at which it was interrupted.[7] This occurs *only* when the termination workspace name was `continue`. If `⎕twsid` was used to specify a different name, that workspace will *not* be loaded.

A `continue` workspace that the system saves for you by either of these automatic mechanisms seems to you to be unlocked, but to any other user the workspace is locked; its password is your current sign-on password. This prevents any other user who may know your user number from gaining access to your interrupted work.

Decision to `)load continue`

The SHARP APL system keeps a record that indicates whether a `continue` workspace has been saved at the abnormal end of an interactive task. This is informally called the flag for a "hot continue." The "hot continue" indicator is kept in the system's current records regarding a user.

SHARP APL sets the "hot continue" flag for your user number when you explicitly ask it to do so (by the command `)continue`) or when your interactive task ends abnormally and the system saves the active workspace. Thus "hot continue" is *on* when the termination workspace name is `continue` and:

- you terminate the task by the command `)continue`

  or

- your line drops (that is, there is a failure of the communications link from the terminal of a T-task)

  or

- the controlling task retracts the offer to share the name that controls an S-task

  or

- the task is "bounced" by a command issued by the system administrators or by `⎕bounce` executed by the task owner

  or

[7] The resumption works well when you start the new session from the same sort of terminal, and the work you were doing did not involve files or shared variables. At a different terminal, some terminal-specific settings may no longer be appropriate. If you were using files or shared variables at the time `continue` was saved, when you resume, you or your program must take steps to resume their use and to check that they are in step with the workspace.

- the SHARP APL system crashes.

At the start of a new task, SHARP APL attempts to load your `continue` workspace when:

- the task you are starting is interactive (that is, a T-task or an S-task)

  and

- the "hot continue" flag is set

  and

- you are *not* a constrained user.

It is possible that, when SHARP APL attempts to load `continue`, it finds that the workspace no longer exists. That might happen when, in the interval since `continue` was saved, another task dropped it or (rarely) because the recovery procedure following a crash was unable to recover it. In such a case, at your next sign-on of an interactive task, you see the message `ws not found`.

The "hot continue" flag is set to *off* (and thus a subsequent interactive task will not start by loading `continue`) when:

- the system automatically loads `continue` at the start of an interactive task

  or

- you terminate a task with `)off`

  or

- you save `continue` by the command `)save continue`.

(In the last two cases, the "hot continue" flag is probably *off* already. However, making sure it is *off* is not completely redundant: there is the possibility that in the interval another interactive task on your user number has ended abnormally and has thus set the "hot continue" flag *on*.)

A Task's Access to Saved Workspaces

Each user number has a *library* of saved workspaces and also a library of APL files. The full name of a workspace or file consists of its library number followed by an arbitrary name of up to 11 characters. You can leave out the library number when you are referring to a workspace or file in your own library.

A task has unlimited access to the workspaces in its owner's library. A task can load or copy a saved workspace that belongs to the private library

of other user *only* by knowing the user number, the name of the workspace, and its password (if its owner has set one).

The command `)lib` gives you a list of the saved workspaces in your own library or in a particular public library. However, SHARP APL will not tell you the names of workspaces in the private library of another user. Thus, you can make use of a workspace from someone else's library only when you have other means to know about it – for example, its owner told you the user number, workspace name, and (if need be) password.

A task can *load* a saved workspace. Doing so discards the entire current contents of the task's active workspace (with the exception of the session pronouns `□sp` and `□ht`) and replaces it with the saved workspace. A task may also *copy* all or part of the contents of a saved workspace without discarding the current active workspace. (See the commands `)load` and `)copy` in Chapter 17, "System Commands".) As with any system command, these commands may be issued only from an interactive task (S-task or T-task).

You cannot save a workspace into the private library of another user. Thus even when you are able to use such a workspace, you have what amounts to *read-only* access to it, since you cannot save it back into the library it came from.

A workspace may be *locked*, that is, may require a password. Only the workspace's owner can set the password. You must include the password in the command that loads, copies, or drops such a workspace. When a password has been set, it is required of all users, even the owner. The SHARP APL system keeps a record of unsuccessful attempts to load or copy a locked workspace.

A Task's Access to Files

SHARP APL files are grouped into libraries in the same way as workspaces. Like a workspace, the full name of a file consists of a user number followed by a name. (See Chapter 14, "APL Component Files" for a discussion of the APL component files managed by SHARP APL.)

Your access to a file is determined by the file's *access table*. When a file is created, its access table is empty. The user whose task created the file is called its *owner*. An empty access table permits the owner to have complete access to the file, and denies all access to other users.

By putting numbers into the access table, a file's owner may specify what level of access various users may have to it. The owner of a file may grant specific rights to tasks belonging to a particular user number. Permission is stated as three numbers: a user number, a code indicating what that user

can do, and, optionally, a numeric password that the user must include as part of a sentence that refers to the file. There is also provision to grant a permission to "anyone" by using `0` instead of a specific user number.

When your user number appears in a file's access table, you have the level of access therein granted to you. This is true even for a file in your own library; you may if you wish exclude yourself from some kinds of access to your own file. If this happens by mistake, the system administrators can assist you in regaining access to your affected file.

The APL component file system is described in Chapter 14, "APL Component Files", and the various system verbs for access to files in Chapter 16, "System Nouns and Verbs".

Starting a T-Task

A T-task is the only type of task directly controlled by a human being. You start and control a T-task from a terminal that is linked to the SHARP APL system.

Depending on how the terminal is connected, you may need to go through some preliminary steps that establish the terminal's transmission speed and character encoding. If you are connected through a network, you may have to identify the SHARP APL system to which you seek a connection. Your call may be routed to a *session handler* appropriate to the device you are using. Details of these preliminary steps are outside the scope of this manual; you must consult the guide for the particular system and device you are using.

Sign-on

When your terminal makes contact with the SHARP APL system, the system solicits a sign-on by sending you a *blot*. This is a place in which to write your user number and password so that it is not visible to others. At a printing terminal a blot is formed by overstriking a zone about 35 characters wide. At some video display units (VDU), a blot is simply a field in which what you type is invisible.[8]

At the blot, you type your sign-on. It consists of a right parenthesis, your user number, a colon, and your password (shown below as *x0b8dqha*):

)	1234567	:	*x0b8dqha*

[8] The SHARP APL system may be unable to distinguish certain VDUs from printing terminals, and hence may transmit a succession of characters to a VDU as if it were a typewriter. Depending on the VDU, this may or may not conceal your sign-on, so you may need other precautions to keep it unobserved.

The initial parenthesis serves to indicate that what follows is a sign-on. It also permits the SHARP APL system to distinguish between various ASCII and APL conventions, since they use different transmission codes for that character. When the system identifies your terminal as one capable of sending and receiving APL characters, it interprets what you type accordingly and may send you APL characters. When the system identifies your terminal as an ASCII-only device, it does not expect you to send APL characters and does not attempt to send you APL characters that are not in the standard ASCII set.

Optionally, you may at the same time revise your password, by writing another colon and the new password on the same line with the user number and the present password. For example, the following would sign on with password *x0b8dqha* and change your password for subsequent sign-ons to *Arrgh6*:

)	1234567	:	*x0b8dqha*	:	*Arrgh6*

The SHARP APL system acknowledges your sign-on and announces the new task. (Optionally, this may be preceded by a brief message from the system administrator, as is visible in Figure 2–1.) The task announcement displays (on a single line):

- ***Task number.*** An arbitrary number between 1000 and 9999 is assigned to each task as it starts. No two tasks have the same number at the same time.
- ***Date.*** The date is written in ISO Standard format, YYYY–MM–DD. The time appears as HH:MM:SS. Time is shown in the time zone adopted for the central computer, regardless of the location of the terminal (so that all users and records use a common time system regardless of location).
- ***User Name.*** An arbitrary sequence of up to 11 characters associated with your user number at the time it was established. This name may not include embedded blanks. The first three letters of this name may be used by the *)port* command (described in Chapter 17, "System Commands").

Typically, on a line below, the name of the timesharing system is displayed.

```
)[illegible]
opr: maintenance scheduled Nov 07 02:00-05:00utc.
  9232) 1988-11-05 21:43:04 pcberry

sharp apl service
```

Figure 2–1 : Sign-on of an interactive task.

If your user number is constrained, none of this information is displayed.

Automatic Workspace Load

When a T-task starts, it may automatically load a workspace. There are two circumstances that cause this to happen:

Continue The system automatically loads your continue workspace when:

- your previous interactive task (that is, T-task or S-task) was terminated by a line drop (communications failure) or was forced off by □*bounce*

  and

- the task's active workspace was saved under the name *continue*

  and

- you are *not* a constrained user.

When SHARP APL loads *continue* in this fashion, it displays the message *saved* followed by the date and time at which the workspace was saved.

Profile The system automatically loads your profile workspace. (This workspace may be named *Profile*; however, the system administrators have the option to designate a different name for profile workspaces.) SHARP APL automatically loads your profile workspace when:

- you are a constrained user

  or

- you are a profiled user *and* you have no "hot continue." To load the profile workspace, SHARP APL uses □*qload*, so that there is no explicit acknowledgment that a workspace has been loaded and no display

of the workspace's timestamp. As it usually does when it loads a workspace, the system automatically executes the latent expression `⎕1x`, which in turn starts the application for you. (See the discussions of `⎕qload` and `⎕1x` in Chapter 16, "System Nouns and Verbs".)

Immediate Execution

Immediate execution works as follows:

At the end of each display, except display produced by ⍞, the SHARP APL session handler transmits a *newline* sequence (normally, carriage return and line feed) to move the cursor to the beginning of the next line. Then it prompts for input by transmitting six blanks. It also unlocks the keyboard or sounds the bell, depending on characteristics of the terminal.

The terminal sends one line of instruction to the SHARP APL system. Pressing *Enter* marks the end of that line of input from the terminal. As soon as the SHARP APL interpreter receives the completed line, it acts upon it. When the interpreter has completed the indicated action, it displays the result (if any), followed by *newline*. Then it again transmits the six-blank prompt and waits for a new transmission from the terminal. ("Completed the indicated action" may, of course, be anything from a trivial or even empty request up to a major job lasting minutes or even hours or days.)

The action requested from the interpreter may itself solicit input from the terminal. By doing so, it puts the session temporarily into a different *mode* of input. The next transmission from the terminal is treated as data to be accepted, or as an editing instruction, rather than as a statement in the APL language to be executed.[9]

Regardless of mode, every line of input from the terminal requires a response from the SHARP APL system. Moreover, the system will not accept further lines until it signals that it has completed processing of the previous line.[10]

The T-task continues until the terminal sends a sign-off command or until the task is bounced or terminated because of a communications or system failure.

Input Modes

The possible modes of interaction at a terminal are:

9 One reason for electing to have a user number marked as profiled or constrained is to help a naïve user to start a session without having to invoke an application explicitly. This is done on the assumption that such a user will not know what to do unless prompted by the program, although user convenience is another good justification.

10 Thus the SHARP APL system does not itself support buffered or "type-ahead" input, although the network, session handler, or terminal may.

Immediate execution This is the default, as described in the preceding section.

When the first non-blank character is) (right parenthesis), the line is interpreted as a system command. See Chapter 17, "System Commands". When the first non-blank character is ∇, the line is interpreted as a call to the ∇-editor. See Chapter 8, "Verb Formation".

Character input In response to the symbol ⍞, the system unlocks the keyboard (or sounds the bell) without moving the cursor. The next line that you type is interpreted as character data and is returned as the value of the symbol ⍞. See Chapter 4, "Nouns and Pronouns".

Evaluated input In response to the symbol ⎕, the system displays the symbols ⎕: on a new line at the left margin, and then moves to the next line and indents by six blanks. Then it unlocks the keyboard (or sounds the bell).

The next line you type is treated as a line of APL to be executed. Its result is returned as the value of ⎕. You can write any APL line that would be acceptable in immediate execution mode, including a system command; but you may *not* invoke the ∇-editor and you may not branch to a different line. (However, → alone to abort execution – the so-called "naked branch" – is acceptable and aborts the entire sequence that called for ⎕-input. The input request can also be interrupted by entering an *input interrupt*. See Chapter 9, "Control of Execution" and Chapter 4, "Nouns and Pronouns" for a discussion of both topics.)

Editor input Once you have invoked the ∇-editor, the next line you type is interpreted in the context of the editor. In general, the editor prompts by displaying a line number in brackets. (See Chapter 8, "Verb Formation".)

Starting an S-Task

An S-task is in most respects identical to a T-task, except that it has no terminal. Instead of a communications link over which input is sent and

output received, the S-task is controlled by a name shared with AP1, the APL Auxiliary Processor, described in Chapter 12, "S-tasks".

An S-task can only be started from another task. After sharing a name with AP1,[11] the parent task transmits the same characters as would be used to sign on a terminal. AP1 replies with a blot, and the parent task sends right parenthesis, user number, colon, and password, just as it would to start a T-task.

An S-task, when initiated, starts in the same way as a T-task. That is, if there is a "hot continue" workspace, the system automatically loads it (unless you are a constrained user). If you are a profiled or constrained user, the system automatically loads your profile workspace.

Like a T-task, an S-task continues until signed off or bounced or until it loses its communications link. For an S-task, the communications link is the name that the controlling task shares with AP1. If the controlling task ceases to share that name (because it withdraws the offer to share, clears the workspace, erases the name, or because the controlling task terminates), the S-task terminates, unless *retract permission* was previously established. The system saves its active workspace under the name `continue` and sets the "hot continue" flag in the same way as for a T-task that has lost its communications link. This causes the system to load `continue` automatically at the start of then next interactive task (T-task or S-task).

Starting an Immediate N-Task

An immediate N-task is started by another task. The parent task executes the system verb `⎕run`. The procedure is described in Chapter 13, "Batch Tasks".

An N-task starts in either of two ways:

- It starts with an active workspace identical to the active workspace of its parent task, executing the same instruction. (The instruction should contain a test that differentiates the new task from its parent.)
- It starts by loading a workspace named in the argument to `⎕run` and executes the sentence stored in that workspace's latent expression, `⎕lx`.

In either case, it runs until it completes the sentence thus provided, or until an error occurs, and then terminates.

11 The procedure is described in Chapter 11, "Shared Variables" and Chapter 12, "S-tasks".

Starting a B-Task or a Scheduled N-Task

You do not start a B-task directly. Instead, you submit a request to the Batch Task Scheduler. The request contains essentially the same information as would be required in the argument to `□run`, and additional specifications indicating:

- Whether the task is to be run at a specific time (a *scheduled N-task*) or may be run at the discretion of the Batch Task Scheduler (a *B-task*).
- Constraints on the date and time at which it can be run or a list of jobs that must be completed before it can be run.
- Indication of whether the task can be restarted following interruption:
  - by a normal end to operations (system shutdown)
  - by an abnormal end to operations (system crash).

The details of the procedure for submitting a request to the Batch Task Scheduler are described in a separate document, *Batch Tasks in SHARP APL*.

An N-Task or B-Task Cannot Obtain Interactive Input

A batch task is never in immediate execution mode and never in either of the input modes. When a batch task encounters an untrapped error or a request for one of the input modes, it terminates.

When a batch task terminates, SHARP APL saves the task's active workspace under the name provided in the argument to `□run`. However, if the batch task's last act before terminating was to clear its active workspace, there is nothing to save, and the system does not save anything. (See the discussion of `□trap` in Chapter 10, "Event Handling" and Chapter 16, "System Nouns and Verbs".)

Tasks Belonging to Constrained Users

The system administrators can designate certain user numbers as *constrained*. When you are a constrained user, you can use one or more applications written in APL, but you cannot interact directly with the APL interpreter, and you never see an APL error message.[12]

12 This manual is not addressed to constrained users but may be of use when you prepare a workspaces that a constrained user will use.

When you sign on as a constrained user, the SHARP APL system automatically loads a profile workspace for you.

As soon as it has loaded the designated workspace, the system executes the workspace's latent expression, `⎕lx`, in the same way as for any user. That starts an application for you. The application runs in the usual way. The application may then load some other workspace, and thereby pass control to the latent expression of the workspace thus loaded.

As soon as whatever application is running comes to a halt, the system signs you off. A halt occurs whenever the task returns to immediate execution. That includes immediate execution because

- the application has come to a normal end

  or

- the application has encountered a request for evaluated input (⎕-input)

  or

- the application has encountered an interrupt or error that it has not successfully trapped.

If you are a constrained user, when your interactive task is terminated (for example, by being bounced or by a failure of the communications link), SHARP APL terminates your session without saving a copy of your active workspace. The idea is to prevent a constrained – and presumably unsophisticated – user from being thrust into the middle of a previously interrupted session.

The table in Figure 2–2 summarizes the differences between a standard user, a profiled user, and a constrained user.

EVENT	**APL TASK TYPE**		
	Normal	**Profiled**	**Constrained**
Sign-on	If *hot continue* is off, perform `)clear` and enter immediate execution.	If *hot continue* is off, `⎕load 'Profile'`.	`⎕load 'Profile'`
Sign-on	If *hot continue* is on, `⎕load 'continue'`, and turn off *hot continue.*		`⎕load 'Profile'`
Halt	If not successfully trapped, enter immediate execution.		`)off`
`⎕bounce` or **Line drop**	If `clearout` is set, or if `⎕trap` action `d clear` is set, then `)off`. Otherwise, `)save continue` and turn on *hot continue.*		`)off`

Figure 2–2 : Comparison of User Types.

3 *Grammar*

APL is an *imperative* language; its sentences do not make assertions or ask questions, but issue instructions. In natural language, the subject of an imperative sentence is not stated. Similarly, in APL sentences are addressed to an unidentified interpreter whose task is to carry them out.

APL Syntax

The SHARP APL system executes sentences written in the APL language. To interpret a sentence correctly, the interpreter must observe the language's grammatical rules. You, the human user of APL, need a similar understanding, both so you can interpret sentences you read and so you can write sentences that, when executed, do what you intend. This chapter describes APL's grammar.

Syntactic Classes

The *syntax* of a language is the body of rules that govern the way sentences are assembled. A sentence in APL is put together from names and symbols. They fall into the following syntactic classes:

Noun A noun is made up of numbers, characters, or boxes arranged in an *array*. A noun may also be a package.

Constant A noun whose value occurs directly in a sentence (for example, the numbers `1 2.2 3` or the characters `'abc'`).[1]

Pronoun An arbitrary name that has been assigned to a noun, so that one may refer to the noun by using its name. It is called a *pronoun* because the name is an arbitrary substitute that is understood to refer to the value most recently assigned to it. In mathematics, a pronoun is commonly called a *variable* because the value to which it refers may vary.

Fundamentally, the aim of data processing is to transform the nouns (data) that you have to some other nouns that you find more useful, more interesting, or more convenient. Both the input and the output are – by and large – nouns.

[1] The ISO Standard for APL uses the term *literal* to refer to what is here called a *constant*. "A Dictionary of APL" uses *literal* to refer to a noun whose items are characters rather than numbers or boxes.

Verb A verb[2] specifies an action to be taken. In general, a verb applies to a noun or a pair of nouns and produces a noun as its result. Defining verbs is thus central to data processing: verbs transform the nouns you have to the nouns you would rather have.

The noun to which a verb applies in a particular sentence is called the verb's *argument.* In APL, a verb may take no arguments, one argument noun (to its right) or two argument nouns (one on either side).

Primitive verb A *primitive* verb is one defined in the APL language. It is "built in" to the SHARP APL system, so that its meaning is always known to the interpreter. Each primitive verb is denoted by a single *symbol* such as +. The primitive verbs are described individually in Chapter 6, "Verbs".

System verbs The interpreter also recognizes a set of *system verbs* that denote extensions to the APL language for which no symbol has been designated. Each has a *distinguished name* consisting of the symbol □ followed by certain letters. They are described individually in Chapter 16, "System Nouns and Verbs".

User-defined verb A *user-defined* verb has an arbitrary name rather than a symbol. Its definition consists of a *header* that declares its name and establishes internal names for its arguments and result, and a *body* consisting of lines containing APL sentences that describe how the verb arrives at its result. You use a defined verb in much the same way as a primitive verb; like a primitive verb, it may take either one or two arguments.[3] Creating a user-defined verb in effect extends the APL language. Once you have defined a new verb, you can execute it or make use of it in the definitions of other verbs. Programming is the art of writing definitions for user-defined verbs.

Adverb An *adverb* modifies the action of a verb to produce a new verb. Each adverb is denoted by a symbol. The adverbs are described individually in Chapter 7, "Adverbs and Conjunctions".

Conjunction A *conjunction* modifies the action of a pair of verbs, or a verb and a noun, to produce a new verb. Each conjunction

[2] Also called a *function* or a *program.*

[3] It is also possible to define a verb that takes *no* arguments, and which therefore behaves like a noun; its value is the result returned by executing the verb.

is denoted by a symbol. The conjunctions are described individually in Chapter 7, "Adverbs and Conjunctions".

Copula Like "is" in English, the copula ← links a name to the noun being named. Also called *assignment.*

Branch The *branch arrow* → alters the sequence in which the lines of a user-defined verb are executed. Used from the keyboard or in the recovery expression of `⎕trap`, it may also restart execution of a verb whose execution has been suspended.

Punctuation Instructions written in APL are grouped into phrases, sentences, and paragraphs, just as they are in a natural language. Parentheses are the principal punctuation in APL; as in natural language and in conventional mathematics, they set off a phrase that must be treated as a unit. SHARP APL also uses certain other delimiters.

Evaluation

To evaluate a sentence is to replace each of its adverbs or conjunctions and their arguments with the resulting verb, and then to replace each verb and its arguments with the resulting noun. You end up with a noun (unless the sentence has no result).

The APL interpreter either displays the result (when you did not say what else you wanted done with it) or passes it on to be stored or to be processed by some other verb.

Order of Execution

When a sentence contains just one verb and its arguments, all that is required is to execute the verb. When a sentence contains several verbs, or both verbs and adverbs, you have to have rules to decide what applies to what. From them, you can deduce the order in which to evaluate the various verbs.

You may read an APL sentence from left to right. Each verb (including the derived verbs you get by evaluating an adverb or conjunction) applies to the part of the sentence to its left and the part of the sentence to its right. For example, in the sentence

```
a÷b÷c÷d
```

the first division takes `a` as its left argument, and `b÷c÷d` as its right argument. Within the expression `b÷c÷d`, the divisor of `b` is `c÷d`. Thus:

```
a÷b÷c÷d  ⟺  a÷(b÷(c÷d))
```

Because you are reading from left to right, this arrangement is not symmetric. There is at most one noun to the left, but all the rest of the sentence to the right. Another way to say the same thing is: A verb takes as its arguments the noun immediately to its left but the entire phrase to the right. Some authors express that by saying a verb has *short scope* on the left and *long scope* on the right.

Ruthlessly Consistent Order of Execution

APL's rule for order of execution is applied equally to all verbs, both primitive and user defined. This consistency makes an APL sentence easy to understand.

Nevertheless, the very consistency may surprise you. That is because APL does *not* make exceptions for certain common verbs that in ordinary algebra or arithmetic are given higher precedence than other verbs. For example, the expression

```
a⋆b-c×d
```

is executed in exactly the same order as `a×b×c×d` or `a÷b÷c÷d` (just mentioned). That is, the `⋆` takes as its right argument the expression `b-c×d`, and `-` takes as its argument the expression `c×d`.[4]

Conventional languages use complicated orders of execution, yet are similar to APL when it comes to assignment. Common statements such as `x=b*c` are executed right to left – the multiplication is performed first, followed by the assignment. APL is simpler than other languages in that the right to left execution order is preserved throughout the language.

Infix Notation

In APL, as in arithmetic, a verb appears *between* its arguments. That is, the verb is an *infix*.[5]

When a verb has only one argument, the argument is to the right of the verb.

Ambivalence

Valence is the number of things with which something can combine. In APL, the same verb can be used with two arguments (one on either side) and also with just one argument, on the right. Since the same verb can be used either with one argument or with two, verbs are said to be *ambivalent.*

4 In conventional algebra, the expression would be executed as if it were written `(a⋆b)-(c×d)`.

5 In languages such as C or LISP, primitive verbs are infixes; user-defined verbs are prefixes; in PostScript, all verbs are postfixes.

For example, the symbol – appears twice in the following sentence:

```
-a-b
```

The first – denotes "negation": a quantity is to have its sign reversed. The quantity to be negated is the difference between `a` and `b`. The second – is used with two arguments and means "subtraction."

As this example shows, the practice of using the same symbol with just one argument, or with arguments on both sides, is already familiar from arithmetic.

Niladic, Monadic, and Dyadic Use

When a verb is used with arguments on both sides, it is said to be a *dyadic* use of the verb. Dyadic means "dealing with two arguments."

When a verb is used with only one argument, it is said to be a *monadic* use of the verb.

When a verb is used with no argument, it is said to be a *niladic* use of the verb. Verbs that are niladic can only be used that way – monadic or dyadic uses are forbidden.

The phrase "dyad +" refers to the verb + when it is used dyadically. The phrase "monad +" refers to the verb + when it is used monadically.

Grouping

As in natural language, parentheses surround an expression that is to be treated as a whole. When you evaluate a sentence that contains parentheses, you have to evaluate what is *inside* the parentheses before you can apply to it a verb that is *outside*. For example, the expression

```
(a+b)×(a-b)
```

indicates that the arguments of the verb × are the entire expression `a+b` and the entire expression `a-b`. Thus you have to evaluate both `a+b` and `a-b` before you can start to evaluate ×.

Parentheses are redundant when they have no effect; they have no effect when the order of evaluation they impose is what the system would do anyway without them. Since in APL a verb applies to the entire expression to the right of it, it is never necessary (but not harmful) to put parentheses around a verb's *right* argument. You get the same result from

```
(a+b)×a-b
```

as from

```
(a+b)×(a-b)
```

In SHARP APL, the expression inside a set of parentheses must evaluate to a noun.[6]

Tree Structure of a Sentence

The structure of a sentence can be represented by a tree. For a sentence with a single verb, you depict the verb as the root of the tree and its arguments as the branches.

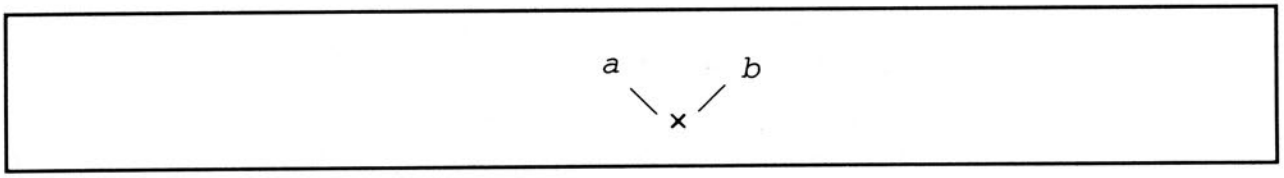


Figure 3–1 : Tree representation of `a×b`.

Where the argument is in parentheses, draw a similar tree to represent the expression inside the parentheses.

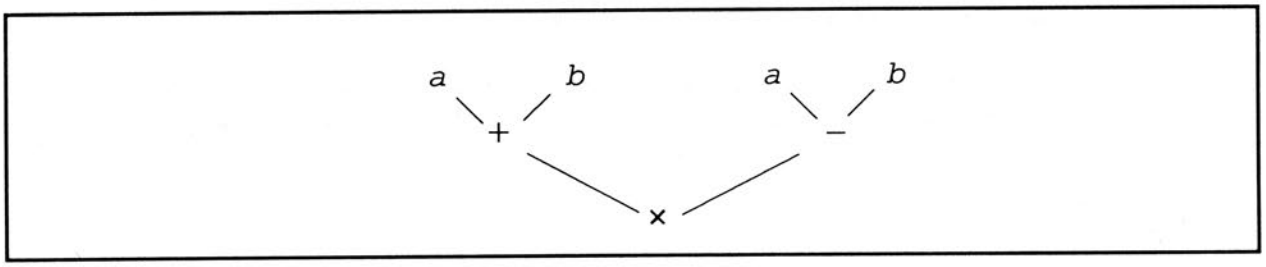


Figure 3–2 : Tree representation of `(a+b)×(a-b)`.

6 To accommodate its use of adverbs and conjunctions, "A Dictionary of APL" permits parentheses around an expression that evaluates to a verb, but SHARP APL does not.

For a sentence confined to verbs and nouns, and which refrains from re-defining the syntactic class of its names in mid-sentence, the tree can be constructed by the following (recursive) procedure:[7]

- Remove redundant parentheses surrounding the sentence (if any).
- If there is no verb, use the remaining noun. EXIT.
- If there is a verb, connect the root verb to its arguments, as follows:
  - The root verb is the first verb (scanning from left to right) that is not inside parentheses.
  - Its left argument is the part of the sentence to the left of it. Its right argument is the part of the sentence to the right of it.
- Apply the same rule to each of the arguments.

Left-to-Right and Leaf-to-Root

When the interpreter evaluates a verb, it has to know the values of the verb's arguments. But a verb applies to the *entire expression* to its right. So the interpreter has to find the value of the entire expression to the right before it can evaluate a verb. (That is true for any interpreter, human or machine: you cannot evaluate the root verb until you have established values for the leaves.)

The fact that APL is read from left to right requires the interpreter, when it executes a sentence containing several clauses, to evaluate the subordinate expressions (to the right) *before* it can interpret the main one (at the left). This is sometimes called the "right-to-left rule."[8] It is perhaps better called "leaf-to-root" execution. Once you remove any redundant parentheses, *the root verb of an APL sentence is the first verb not enclosed in parentheses.*

Precedence of Verbs, Adverbs, and Conjunctions

The precedence rules of a language determine whether some verbs are evaluated ahead of others regardless of their position in the sentence. APL has only one rule of precedence:

- *An adverb or conjunction has higher precedence than a verb.*

This is because an adverb or conjunction modifies a verb, and you cannot execute the verb appropriately until you know how it has been modified.

Among themselves, all verbs have equal precedence; among themselves, all adverbs and conjunctions have equal precedence.

---

7 A listing of APL programs to produce trees in this fashion may be found in Berry et al., *APL and Insight*, paper presented to Colloque APL, Paris, 1971; reprinted by APL Press, 1974.

8 That name makes it sound somehow backwards, or somehow different from ordinary mathematics. In fact this is exactly the standard rule in mathematics.

- *No verb has higher precedence than any other verb. That includes both primitive verbs and user-defined verbs.*
- *No adverb or conjunction has higher precedence than any other adverb or conjunction.*

Because all verbs have equal precedence, the *only* determinants of order of execution are parentheses (in the usual way) and position in the sentence.

Conventional algebra also gives equal precedence to arbitrary functions. However, it has special rules for a few common functions. It gives exponentiation higher precedence than multiplication or division, and multiplication or division higher precedence than addition or subtraction.[9] APL does *not* retain these precedence exceptions and is therefore easier to read and use than C-like languages.

Precedence in a Sentence with Adverbs or Conjunctions

An adverb takes as its argument the noun or verb that it follows.

A conjunction takes as its arguments the verbs (or the verb and noun) on either side of it.

Because adverbs and conjunctions have higher precedence than verbs, when (while reading through a sentence) you come to something that is modified by an adverb or conjunction, you evaluate the adverb or conjunction at once and replace it (and its arguments) by the resulting derived verb. For example, in the sentence

```
(a+b) ×⍤(1,r) -(c+d)
```

the root verb is *times-rank-l,r.* Reading left to right, the quantity `(a+b)` is to be multiplied. But × is modified by the rank conjunction ⍤, so you have to evaluate the rank conjunction before you can do anything about the multiplication. The conjunction's arguments are the verb × on the left and the noun `(1,r)` on the right.

In order to evaluate ⍤, you have to evaluate its argument `(1,r)`. That done, you establish that the root verb is *times-rank-l,r.* Its right argument is:

```
-(c+d)
```

Notice that although the right end of the sentence contains the characters

[9] The precedence rules of conventional algebra are reflected in its typographical conventions. The higher-precedence functions are conventionally denoted by juxtaposition (for instance, to indicate multiplication) or by a combination of juxtaposition and super- or sub-scripting (for instance, to indicate exponentiation or indexing). However, when computer programmers abandoned one-letter names, they had to abandon juxtaposition too. To simplify work from terminals, APL and most other systems have *linearized* their input, and thus abandoned subscripts and superscripts. Languages such as Fortran and BASIC have retained the conventional precedence even though they lack the typographical conventions that in mathematics help express precedence.

```
(l,r)-(c+d)
```

`(c+d)` is *not* subtracted from `(l,r)`. The precedence rule means that `(l,r)` is the argument of the conjunction ⍤ and gets evaluated before the − is performed.

Notice too that although a verb rarely requires parentheses around its right argument, a conjunction may indeed require them. In the foregoing example, omitting the parentheses around `l,r`

```
(a+b) ×⍤l,r-(c+d)
```

makes `l` the argument of the rank conjunction ⍤. The sentence then reads

the quantity `(a+b)` *times-rank-*`1` *the quantity* `,r-(c+d)`

which is not the same thing at all.

Other Mechanisms of Grouping

Although parentheses are the principal mechanism for grouping within a sentence, there are others that affect both execution within a sentence and the overall grouping of lines within a definition. They are discussed next.

Brackets[10] are anomalous in APL. So are semicolons. Both date back to the earliest APL implementations. Both are obsolescent. That is, as the language develops, other more consistent mechanisms become available for tasks where brackets or semicolons were formerly indispensable. However, for the sake of consistency with old programs, the old meanings of `[]` and `;` are retained, even though need for them has vanished in newer dialects of APL, such as J.[11]

Brackets Group in the Same Way as Parentheses

The square brackets `[]` serve both to group the expression inside (in much the same way as parentheses) and also to modify whatever is to the left of the bracketed expression. For example, in an expression such as

```
x[a+b]
```

the value found by evaluating `a+b` is used to select particular members of the list `x`. The selection index is the result of the entire expression within the brackets.

10 That is, the square brackets [ ], in contrast to parentheses ( ), which are curved.

11 J is a new dialect of APL, available as shareware. J has removed the anomalous features of traditional APL. It also uses the ASCII character set instead of the APL character set and so is easy to use in conjunction with a wide range of hardware and software. J is described in "APL\?", APL90 Conference Proceedings, *APL Quote Quad*, vol. 20, no. 4., 1990.

Brackets are used to denote the verb *selection by indexing* (See "Bracket Indexing" in Chapter 6, "Verbs") and certain cases of the rank conjunction (described in Chapter 7, "Adverbs and Conjunctions").

Semicolon Is a Separator

The semicolon ; is used as a separator. There are two contexts in which it may be used:

- inside brackets used for indexing an array of two or more axes
- separating expressions in side-by-side output.

Semicolons inside brackets. Inside brackets, semicolons separate the expressions that specify positions along the various axes of an array. Thus one semicolon is required for an array that has two axes, two semicolons for an array that has three axes, and so on. Each of the expressions thus separated needs no other delimiter; in particular, it never needs to be enclosed in parentheses.

As long as brackets are used for indexing, semicolons appear to be a necessary adjunct to them. The direction of evolution in APL is to provide superior alternatives for bracket-semicolon indexing without actually immediately ceasing to support it.

Semicolons on mixed output. Anywhere else, semicolons separate successive sentences for side-by-side output to a terminal. For example, the sentence

```
'f(x) when i=';i;': ';table←(i←24) f x
```

sets the value of `i` to `24`, and evaluates `i f x` and stores it as `table`. Then it displays the message

```
f(x) when i=24:
```

with the value of `table` to the right of it. This use of semicolon was formerly called *mixed output* because (since the various arrays that are displayed are produced by separate sentences) there is no need for them to be of the same type or shape. In modern APL, the same effect can be produced in a more consistent manner by the verb ⊃ (*link*). The direction of evolution is to discard semicolons in mixed output. They are not mentioned in the ISO standard and have been dropped from several APL systems, although not (at this point) from SHARP APL.

Mixed output with semicolons is permitted *only* at the root level of the sentence. A set of sentences separated by one or more semicolons may *not* be assigned a name and may *not* used as the argument of any verb. For example,

```
x←(a;b)
```

is rejected as a *syntax error*, while

```
x←a;b
```

has the same effect as

```
x←a
a;b
```

A set of phrases separated by semicolons may be enclosed in parentheses *only* when the parentheses surround the entire sentence (and are therefore redundant). For example, the outer parentheses in the following are permissible but pointless:

```
('a=';a;' b=';b;' a foo b=';a foo b)
```

Other Punctuation

No other punctuation affects the interpretation of an APL sentence. The symbols

: (colon)

◇ (diamond)

⍝ (lamp)

delimit parts of a *line*, but do not occur within a sentence except as they may be included in a character constant. (See "Scanning the Line for Quotes and Comments" and "Dividing the Line into Sentences," later in this chapter.)

The colon occurs only to mark a label on a line in a user-defined verb and is not part of an APL sentence.[12]

Similarly, the quote mark ' delimits a character constant within a sentence. However, as will be explained in the discussion of "Analysis of an Input Line", you have to resolve the use of quotes for the entire line before you can break it into sentences.

Thus, apart from the anomalous use of brackets and semicolon, grouping with parentheses is the only form of punctuation within an APL sentence.

Colon and Label

A label is a name assigned to a line within the definition of a defined verb. (See the discussion of labels in the section on "Localization of Names" in Chapter 8, "Verb Formation".)

12 In the method of representing a verb called *direct definition*, a colon separates the name of a verb from its definition. That is a typographical convention of the writer, or a convention of a program such as *def* (illustrated in Chapter 1, "Overview"), but not a convention of SHARP APL itself.

The colon : serves to identify a label. A label is a name formed following the same lexical rules as for names of nouns or verbs. If a line is labeled, the line must begin with the label, followed immediately by a colon. That is the only use of the colon. Since there may be at most one label on a line, there may be at most one colon.

Note that a label is attached to a line, not to a sentence. Thus, a line containing several sentences separated by diamonds may nevertheless contain only one label, and the label must appear first on the line.

Lamp and Comment

The lamp ⍝ marks the start of a *comment*, which is a portion of a line that contains text intended for the illumination of the human reader but not for execution by the APL interpreter. As it scans a line from left to right, when the interpreter reaches a ⍝ (other than one embedded within quotes) it ignores the ⍝ and everything to the right of the ⍝ on that line.

Note that a comment is part of a *line* rather than part of a *sentence*. When several sentences appear on the same line (separated by diamonds) each sentence may *not* have its own comment, since the first comment causes the interpreter to disregard the remaining characters on that line.

Since ⍝ ensures that the remainder of the line will not be executed, it does not matter whether the comment contains unmatched quotes, parentheses, brackets, or additional comment symbols. Nor does anything written in the comment affect the ∇-editor, which might otherwise respond to bracketed line numbers or to the symbols ∇ or ⍢.

Arrays

In general, a noun is an *array*,[13] composed of any number of *items*, arranged along any number of *axes* or *dimensions*.[14]

There is no special procedure for saying "This is an array," since every noun is by default an array. An array that contains only one item and has no axes at all is just a particular case of "array," sometimes called a scalar.

When a verb refers to an array, it refers to the array *as a whole*. Depending on what the verb is and how it is used (and how it may be modified by adverbs), the verb either treats the entire array collectively or considers the array to be a set of cells to which the verb applies in *parallel fashion*.

13 The only exception is a *package*, a noun whose various members are identified by name rather than by position in an array structure.

14 SHARP APL limits an array to a maximum of 63 axes.

Types of Nouns

An APL array may be of one of three types. All the items within an array must be of the same type.[15] The three types are:

- numeric
- character
- box.

A box is a single item that is an encoding of an array; the array is said to be *enclosed* within it.

No Declarations

There is no provision in APL to declare in advance the type, shape, or size of a noun. The SHARP APL interpreter simply notices what each noun's type and shape happen to be and allocates it space *as required.* It chooses various internal forms for the storage of numbers but converts between them automatically. Which internal type the system uses to store numeric data should concern you only insofar as it affects the amount of space that a stored array occupies, or as it affects the performance of primitive operations on that data.

The name assigned to a noun may be reused at will and may be assigned to a noun of different type or shape. Thus it is acceptable to say `x←6` in one sentence and later say `x←'Friday'`.

Analysis of an Input Line

The syntactic rules within a sentence are quite simple. Entries that you type from the keyboard are executed a *line* at a time. Interpretation does not start until you signal that the line is complete by pressing the *Enter* or *Return* key, hereafter called *newline.* Similarly, the interpreter executes the definition of a user-defined verb a line at a time. It takes a little more analysis to process an entire line because the line may contain a label or a comment and may be divided into separate sentences by the *diamond* sentence separator. And any of these may occur inside quotes, where they do not count, so the location of quotes and comments must be analyzed before anything else.

15 This restriction is a matter of expediency imposed to simplify the implementation and is not inherent in the APL language.

Preliminary Checks for) and ∇

A line entered from the keyboard may be a *system command.* When the first non-blank character in a line is) (the right parenthesis), SHARP APL treats the entire line as a system command (and gives it no further consideration as APL sentences).

A line entered from the keyboard may invoke the ∇-editor. When the first non-blank character in a line is ∇ (*del*), SHARP APL treats the entire line as an invocation of the ∇-editor (and gives it no further consideration as APL sentences).

Neither a system command nor a call to the ∇-editor may be part of a user-defined verb.

A line may consist of one or more sentences separated by ⋄, the *diamond.* However, a diamond that is part of a character constant or a comment does not act as a separator.

Scanning the Line for Quotes and Comments

Two symbols require special attention early in the analysis of a line. They are the *quote* symbol ' and the *comment* symbol ⍝. Their meanings are as follows:

' ' ***Quotes.*** A character constant is delimited by the symbol ' before it and another ' following it. All the characters between the quotes (but not the delimiting quotes themselves) are part of the constant. Within a quotation, two consecutive quotes denote the quote character itself.

⍝ ***Comment.*** The first occurrence of ⍝ that is *not* part of a character constant (and thus is *not* enclosed within quotes) indicates the start of a comment. All the rest of the line is the comment and has no effect on the interpretation of the line.

Since the character ⋄ between quotes or to the right of ⍝ is simply a character and has no power as a delimiter, you cannot divide a line into sentences until you have scanned for quotes and comments. Proceed as follows:

- Note the locations of the symbols ' and ⍝.
- Scan left-to-right for the first ⍝ that has an even number of quotes to the left of it. (Zero is an even number.) That lamp and everything to the right of it is a comment. Discard the comment. It is not an executable part of the line.
- Count the number of quotes that remain. In a line that has an odd number of quotes, it is impossible to tell which characters are supposed

to be inside and which outside the quotation. Give up. Reject the entire line as a *syntax error*.

- Scanning from left to right, mark each character constant. A character constant starts with the first use of a quote and runs until a closing quote. At a position in the sentence where a character constant has started but not yet ended, two consecutive quotes denote the quote character itself and do *not* serve as a closing quote. For example, the sentence

```
a←'I can''t',b
```

contains a single character constant composed of the seven characters

```
I can't
```

- Any ⋄ that is *not* inside quotes is a sentence separator.

Dividing the Line into Sentences

Each line commonly contains one sentence. Frequently, *sentence* and *line* are equivalent.[16] When a line contains more than one sentence, the successive sentences are separated by the symbol ⋄ (*diamond*).

The sentences in a line are executed in order from left to right. When one of the sentences is a *branch*, it may cause control to go immediately to the destination of the branch; if it does so, sentences further to the right are not executed.

A sentence that contains no characters, or only blanks, is said to be empty. It does nothing, but it is not an error.

Types of Sentences

APL has three types of sentences:

Branch The first non-blank character is →. It causes control to pass to the line identified by the value to the right of the arrow. Branching is discussed in Chapter 9, "Control of Execution".

Assignment The root of the sentence is the copula ←. The noun resulting from evaluating the expression to the right of the arrow is assigned the name to the left of the arrow. There is no display.

Everything else The result of the root verb is displayed.[17]

16 "A Dictionary of APL" does not include diamond and assumes that *line* and *sentence* are always equivalent.

17 If it has a result. A few primitives return no result; a user-defined verb may be defined so that it has no result.

Dividing a Sentence into Words

A *word* is one or more characters that, in the interpretation of a sentence, can be treated as a unit. For example, each of the APL symbols for a primitive verb or adverb is a single word, since each stands alone. However, a *name* is just one word, regardless of the number of characters required to spell it, just as a *number* is a single word, regardless of the number of digits required to represent it.

APL uses many symbols and several punctuation marks. Each of these symbols is a word by itself.[18] When you see one of those symbols, you can treat that character as a word.[19] You know that it cannot be part of some other word.

Multi-character Words

Words of more than one character occur in three classes:

Name The name assigned to a user-defined noun or verb may contain up to 77 characters. (The characters permissible in a name are described later in this chapter.)

Numeric constant Several digits, and also the decimal point, the negative sign, and the letters `e` and `j`, may occur in the spelling of a single number. The macron `¯` is the negative sign; note that this "high minus" is *not* the same thing as the subtraction sign. A list of several numbers separated by blanks forms a single word.

Character constant A character constant, as described earlier, is delimited by `'` at either end. Between the delimiting quotes, the constant may contain any non-control characters (subject to the limitations of the device to enter or display them).

Blanks

A blank cannot occur as part of a name or part of the representation of a single number. Blanks serve to separate two characters that might otherwise appear to be part of the same number or the same name. Blanks are not otherwise significant, except in character constants.

---

18 For this purpose, we consider not only symbols (such as + − × ÷) to be words, but also any of the punctuation marks (such as ( ) [ ] etc.), even though in English, the category "word" does not usually include punctuation.

19 Except, of course, when it is inside a quote or comment.

Blanks are required to separate two consecutive names (for example, the name of a verb and its argument or the names of two user-defined verbs) or a name and number.[20]

Scanning Strategy for Locating Words

Each of the APL primitives is represented by a single symbol and is a single word. Because it is a single word, it needs no delimiter beside it. In scanning a line of APL, the following algorithm exploits that fact. The strategy is:

- Locate the left-most character that may be the start of a multi-character word. You have found the start of a possible word when:
  - the present character could be the start of a name or a constant
    and
  - the preceding character could *not* be part of a name or constant.
- Assume that the name or constant continues until you reach a character that could *not* be part of it. Then you know that the preceding character was the end of the name or constant.

Numeric Constants

A sequence of characters may spell a numeric constant. A numeric constant may represent just one number or several numbers separated by blanks. For example, the expression

```
b+3.1×a
```

has a numeric constant consisting of just one number (`3.1`), while the expression

```
b+3.1 ¯2×a
```

contains a single numeric constant consisting of the two numbers `3.1` and `¯2`.

Blanks that are surrounded by numbers delimit the numbers from each other but do not mark the end of a numeric constant. (For example, the blank between the `1` and the `¯`.)

In a numeric constant, the first character of the first number must follow a character that could not be part of a constant (or must be at the start of the

[20] In the SHARP APL system, if (while writing the definition of a verb) you put extra blanks between words (perhaps intending to give a more pleasing visual effect), the extra blanks are not retained in the definition. When the interpreter "fixes" the definition (for example, by using the `□fx` verb), the definition is converted to an internal form that does not need delimiters. When you subsequently display the verb's definition, the interpreter regenerates the display from the internal form, but it has no record of the redundant blanks.

sentence). In the preceding example, the numeric constant starts after the symbol +.

Scanning from left to right within the same numeric constant, the first character of each subsequent number must have a blank in front of it. The blank in front of ¯2 is necessary to separate ¯2 from 3.1; however 3.1 ¯2 is a single word.

Complex and Exponential Number Formats

A number may be real or complex. A complex number is written as two real numbers separated by the letter *j*; for example:

```
3.1j5
```

Thus the letter *j* may occur in a number, but not more than once, and only with a real number both before and after it.

A real number may be written in either of two ways: decimal or exponential. The exponential form consists of a decimal number, the letter *e*, and an integer; for example:

```
3.1e¯9
```

Thus the letter *e* may occur in a number, but not more than once (or more than once in each part of a complex number), and only with a real number in front of it and an integer after it.

A number must start with one of the characters 0 through 9 or the negative sign ¯ or the decimal point. After a number's first character, subsequent characters may be any of the characters 0 through 9 or the decimal point. However, a number cannot contain more than one decimal point if real, or one in the real part and one in the imaginary part if complex.

A number may not contain a comma; comma is a stand-alone symbol and thus a word in itself.

Dot is a decimal point when it occurs next to a digit, but a conjunction otherwise.

List Constants

A constant may be a *list* of several numbers or characters. The general rule is as follows: while scanning a line from left to right, upon first encountering the first character of a number, assume that it may be the start of a numeric list. Scan rightward at two levels:

- Look for a blank or a character that could not be part of a number.[21] Since it could not be part of a number, you know that the preceding character is the last of this number.
- Look for a character that could not be part of *any* number. It terminates this list of numbers.

Similarly, a character constant forms a single word. However, the two-level scan is not necessary because (unlike a numeric list) a character constant has an explicit delimiter, the quote mark.

After detecting the opening ', assume that it may be the start of a character list. Scan rightward looking for a closing quote.[22]

A character list is a single word.[23] Anything adjacent to a character list cannot be in the same word. Hence a character constant does *not* need any additional delimiter in front of the opening quote or after the closing quote.

Names

A sequence of characters may spell a name. The first character of a name must be located at the beginning of the sentence or after a character that could not be part of a name.

The first character of a name must be a member of the set considered *alphabetic* (see below) or the symbol quad (⎕). Names that start with ⎕ are called *distinguished names*. The only valid distinguished names are those recognized by the SHARP APL system; that is, although you are free to coin arbitrary names that start with something other than ⎕, you are not allowed to coin arbitrary distinguished names.

The "alphabetic characters" are those permitted in as the first character in user-defined names. The set accepted by SHARP APL contains the 52 characters used in English words (without diacritical marks):[24]

```
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

SHARP APL's set of alphabetic characters also includes the characters Δ and ∆; they are permitted in names in the same way as other alphabetic

21 As you scan, you have to revise the list of characters that could be part of a number. For example, when you have only seen numerals, the decimal point is an eligible character. However, once you have observed one decimal point in a number, a decimal point is no longer eligible to continue the same number.

22 This requires looking ahead for two characters, since once a quote has started, the sequence '' denotes a single quote character rather than the end of the quotation.

23 Note that *word* is here used in a special sense; the list `'hot dog'` is for this purpose one word, even though the blank in the middle means that in natural language we would treat it as two.

24 "A Dictionary of APL" describes the characters used in names as those in the system's "native alphabet," which in principle permits a system to recognize whatever characters are employed in a particular national alphabet. SHARP APL nevertheless confines itself to the 26 Latin-derived characters without diacritical embellishments. You can write characters such as Å and ö in text but not in the names of APL objects.

characters. The characters ⍺ and ⍵ may be used as names but may not form part of a longer name. By convention, they are used as the names for the left and right arguments of user-defined verbs.

Names are not permitted to start with the characters `sΔ` or `tΔ`. These prefixes are reserved for control over trace and stop capabilities.

A name continues until (scanning from left to right) you reach a character that could not be part of it. After its first character, a name may contain any letter of the alphabetic set or a digit. A digit is one of the characters `0` through `9`. A name may not contain a blank. A name may not contain more than 77 characters.[25] Figure 3–3 provides a summary of characters permitted in names and numeric constants.

Permitted in a ***Name***	Permitted in a ***Numeric Constant***

`a ... z`
`A ... Z`
`Δ Δ`
 	
`0 ... 9`
(not at start)
 	
(Also ⍺ or ⍵ alone)
	`0123456789`
	
	`¯ . e j`

Figure 3–3 : Characters permitted in names and numeric constants.

Assigning Words in a Sentence to Syntactic Classes

To execute a sentence, you have to know what applies to what. Such analysis is called *parsing*. Identifying which verbs take which arguments in effect defines the order in which the verbs must be executed. Parsing requires identifying the syntactic class to which each word belongs. The syntactic class of a symbol can be determined from the chart shown in Figure 3–4. The syntactic class of a name is arbitrary; the interpreter determines that by consulting the name's entry in the workspace's symbol table.

[25] SHARP APL applies stricter rules to the names of files and workspaces.

Punctuation	Branch	Copula	Verb	Adverb	Conjunction	Noun
( ) [ ] ; ' ⍝ ◇ :	→	←	+ × − ÷ ⋆ ⍟ ⌈ ⌊ ⊥ ⊤ ⍋ ⍒ < ≤ = ≥ > ≠ ∧ ∨ ⍲ ⍱ ∈ ⍴ ↑ ↓ ∼ ⍳ ○ ? ≡ ⍉ ⊖ ⌽ ⍋ ⍒ | , ⍪ ⊆ ! ⌹ ⊃ ⊂ ∩ ∪ ⊢ ⊣ ⌶	⌿ / ⍀ \	⍤ ⍣ ⍥ . See note, below.	⍺ ⍵ ⍬ ⎕ ∘ See note, below.

Figure 3–4 : Symbols in syntactic classes.

Notes:

. The symbol . denotes a conjunction only when it is *not* part of a number. It is part of a number when the character on either side of it is a numeral.

⊂ ∩ ∪ The symbols ⊂ ∩ ∪ are parsed as verbs even though no use has been defined for them.

∘ In SHARP APL the symbol ∘ occurs only as part of the *dot* conjunction and so is a special case.[26]

∇ ⍢ The symbols ∇ and ⍢ invoke the ∇-editor. If they occur in an APL sentence (except as part of a character constant), they are treated as undefined conjunctions.

The rules governing the use of these classes define the *syntax* of the APL language. In subsequent chapters, members of the various classes are described separately.

26 In "A Dictionary of APL" it is treated as a noun.

4 *Nouns and Pronouns*

A noun is a collection of items of data – numbers or characters – treated together as a whole. For example, in the sentence

```
cost←index×+/3 1.4 19
```

the list of numbers `3 1.4 19` is a noun.

A noun may be the *argument* of a verb; that is, the object to which the verb applies. In the foregoing, the noun `3 1.4 19` is the argument of the verb `+/`.

In general, when the interpreter *executes* (that is, *evaluates*) a sentence, it replaces each occurrence of verb-and-arguments by the resulting noun. It continues this process until no verbs are left, and the entire sentence has been reduced to a single noun.[1] Thus, executing a sentence usually requires reducing it to a noun, which is said to be the sentence's *value* (or *result*). When `index` has the value `2`, the result of the sentence illustrated above is `46.8`; that is, `2` times the sum of `3 1.4 19`.

You may assign a name to a noun. When a noun has a name, you may refer to it by its name. However, a noun is not *required* to have a name. In the example, the noun `3 1.4 19` has no name. Nor is there a name for the sum of `3 1.4 19`. However, the noun resulting from multiplying *index* by the sum of `3 1.4 19` is assigned the name `cost`. Thereafter, when you refer to `cost`, the interpreter substitutes for the name `cost` the noun `46.8`. Since a name stands for a noun, in that sense a name is like a *pronoun* in natural language. In this manual, in places where no confusion would result, the terms *noun* and *pronoun* are often used interchangeably, since a pronoun must always be a name for a noun.

Rank and Shape

The items within an array are arranged in a system of axes or *Cartesian coordinates*. The *rank* of an array is the number of axes it has. The simplest array has zero axes and is called an *item* or, in mathematical terminology, a *scalar*.

In principle, an array may have any number of axes; however, as a practical matter, the SHARP APL interpreter permits no more than 63 axes.

[1] Or no noun at all, when the sentence's root verb has no result.

A single item (say, the number `123.45` or the letter `'K'`) has rank 0: it has *no* axes. A *list* (say, the numbers `12 ¯24.3 4.75` or the characters `'Mary Jones'`) has rank 1 because its items are spread along a *single* axis. A *table*, made up of rows and columns, has rank 2. And so on, for any number of axes.

Arrays are *rectangular*. The *shape* of an array is the number of positions along each axis; that is, the *length* of each axis. For example, in a 3-by-4 table (having three rows and four columns), every row has the same number of items (four), and every column has the same number of items (three).

The axes of an array are identified by number. The first axis is by convention "axis 1," the next is "axis 2" and so on.[2]

Reporting Shape and Rank

The monad ⍴, pronounced *rho*,[3] reports the length of each axis of a noun. Its result is a list containing one number for each axis. The value of each number indicates the *length* of the corresponding axis; that is, how many items lie along that axis. For example, when `t` is a table with three rows and four columns:

```
      ⍴t
3 4
```

This is called "shape of `t`." Applying ⍴ again to find the length of the result of `⍴t` (the shape of the shape of `t`) tells you how many items there are in the noun `⍴t`:

```
      ⍴⍴t
2
```

Thus, two consecutive uses of monad ⍴ give you the *rank* of a noun.

Arrays and Cells

You may consider an array as a collection of *cells*. Each cell occupies the noun's *last* so-many axes. The number of axes devoted to a cell is arbitrary, but they have to be at the *end* of the list of axes. Suppose you want each cell to have `k` axes. Then the last `k` axes are said to be *rank-k cells*, or simply *k-cells*.

For example, suppose the noun `a` is created as follows:

---

2 In a context where you have elected 0-origin counting, the first axis is axis 0, the second is axis 1, and so on. See the discussion of *index origin* in Chapter 6, "Verbs". Newer dialects of APL, such as J, use index origin 0 exclusively.

3 *Rho* is the name for the symbol. The verb it denotes may be used either as a monad or as a dyad. As a monad, the verb is often called *shape*; as a dyad, *reshape*. See the discussion of verb names at the start of Chapter 6, "Verbs".

```
      a←2 3 4⍴'abcdefghijklmnopqrstuvwx'

      a
abcd
efgh
ijkl

mnop
qrst
uvwx
```

Then the list `'abcd'` is a 1-cell of `a`. The two separate 3-by-4 tables are 2-cells of `a`, and the entire 2-by-3-by-4 collection is a 3-cell of `a`. Each of the individual letters is a 0-cell of `a`.

These are all valid ways of looking at the noun `a`. Whether you treat `a` as a single 3-cell, or a pair of 2-cells, or a 2-by-3 table of 1-cells, or a 2-by-3-by-4 array of 0-cells, does not depend on anything in `a` itself. How you look at `a` and partition its axes into cells depends on the verb you apply to `a`. It also depends on the way the verb is modified by the rank conjunction ⍤. (See Chapter 7, "Adverbs and Conjunctions".)

The rank conjunction specifies how many of a noun's axes are used to form a cell. The cell axes have to be taken from the *last* of the noun's axes. Suppose `b` is a rank-4 array, and that the shape of `b` is the list `2 3 4 5`. When you decide to partition `b` into cells of rank `k`, in effect you are taking the last `k` items from the list `2 3 4 5` and reserving them for cells.

When you treat the last `k` axes as cells, the noun may have other axes left over. The axes that come *before* the `k` axes you treat as cells are said to be the *frame* axes. The frame axes are the complement of the cell axes. For example, when the shape of `b` is `2 3 4`, then `b` has:

Frame `2 3 4`	relative to cells of rank 0 (each of which is an item)
Frame `2 3`	relative to cells of rank 1 (each of which is a list of shape `4`)
Frame `2`	relative to cells of rank 2 (each of which is a table of shape `3 4`)
An *empty* frame	relative to cells of rank 3 (each of which – there is only one – is the entire 2-by-3-by-4 array).

Notice that an empty frame means that the entire array is treated whole, as a single cell.[4] Figure 4–1 illustrates the alternative ways of partitioning a 2-by-3-by-4 array into cells.

[4] An empty frame has no frame axes. That is not the same thing as a *zero frame* – one of whose axes has length zero (and therefore has *no* cells).

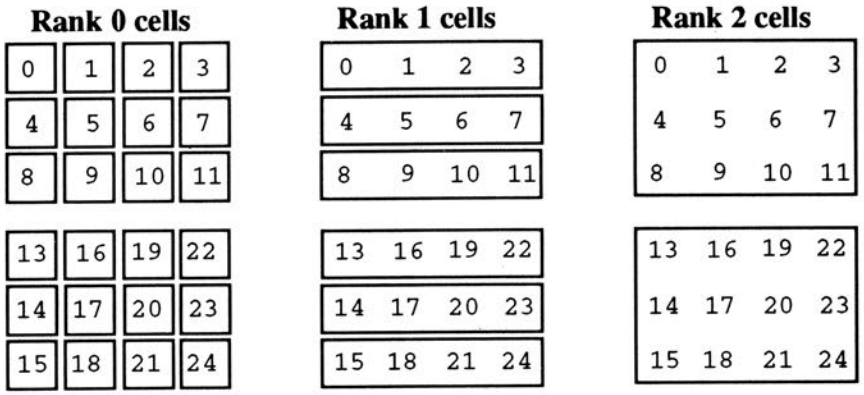


Figure 4–1 : Alternative ways of partitioning a 2-by-3-by-4 array into cells.

The number of cells that a frame contains is the *product* of the noun's *frame-shape*. When any of the frame axes has length 0, the number of cells it contains is zero. Such a frame is said to be a *zero frame*.

Complementary Partition

An array's cell rank is the complement of its frame rank. You can describe the partition by stating:

- the rank of each cell (in which case any leftover leading axes are part of the frame)

  or

- the rank of the frame (in which case any leftover trailing axes are part of each cell).

To describe a partition by specifying the number of frame axes, use a *negative* number. The number of frame axes is the magnitude of that negative number; for example:

Rank `¯1` The first axis is the frame. Each cell has whatever rank remains when the array's first axis is devoted to the frame and all the other axes are cell axes.

Rank `¯2` The first two axes are the frame. Each cell has whatever rank remains.

For example, within the rank-3 array `a` mentioned earlier, the list `'abcd'` may be called a 1-cell of `a` or it may be called a `¯2`-cell of `a`. Similarly, the two 3-by-4 tables within `a` may be called either 2-cells or `¯1`-cells of `a`.

Major Cells

For any array, its ¯1-cells are its *major cells*. The number of major cells is the length of the array's first axis. A rank-0 array (that is, an item, having no axes) has a single major cell, itself.

Infinite Rank

When a verb has *unbounded*, or *infinite*, rank, it treats its entire argument as a single cell, regardless of the number of axes the array may have. Several verbs have infinite argument rank. All user-defined verbs have infinite rank.

SHARP APL does not have a symbol for "infinity." When you need to specify that a verb applies with infinite rank, any large number will do for the rank, provided it is not less than the noun's actual rank.

Types of Noun Items

An item may be:

- a *number*
- a *character*
- a *box*.

Homogeneity of Arrays

All the items in an array must be of the same type: numbers, characters, or boxes.

Within an array of boxes, each box may contain any type of array data (number, character, or box), with no requirement that the various boxes contain data of the same type.

Empty Arrays

An array is *empty* when one or more of its axes has length 0. In principle, the type of an empty array is immaterial. However, the verbs > and ↑ and the derived verb *n*\ are permitted to produce a non-empty array from an empty one, and when they do, the empty array's former type may again become evident.

To facilitate work with character arrays, the empty character list `''`, or any empty array resulting from an operation on a character array, retains its identity as type *character*.

All other empty arrays are numeric. Thus `(⍳0) ≡ 0⍴⍵` for any ⍵, regardless of the type of ⍵ (and even when ⍵ is an array of boxes).

Numbers

During input or output, a number is represented by digits, combined as needed with a decimal point, a negative sign, or the letters `e` or `j`. A negative number has a leading *macron* (or "high minus sign"), as in `¯3.2` or `6.2e¯23`.

A number may be written in any of the following forms:

Integer For example, `2 ¯256`.

Real For example, `2.1 0.009 ¯2.56`.

Exponential For example, `1.6e9 6.2e¯23`.

Complex For example, `3j4.1 1.6e9j¯6.2e¯2`.

In writing the various items in a list of numbers, it is permissible to intermix forms. For example, the following is perfectly valid:

```
x←1 ¯16 2.56 2.08e17 3.1j¯6.3
```

In similar fashion, when the SHARP APL interpreter displays a list of numeric items, it chooses the form independently for each item and thus may also intermix formats.[5]

Although the interpreter permits mixed formats for entry or display, it stores all the members of an array in the same way. The effect of a sentence such as the one just shown is to create an array all of whose items are stored in the most demanding of the forms (see below).

Internal Representations of Numbers

The SHARP APL interpreter uses four different internal forms for the storage of numbers. The forms differ in the space per element they require (so the choice of internal type may have important consequences for storage). The four types are:

Boolean Zero or one. Boolean representation is used for a single 0 or 1 entered from the keyboard, or for the result of a proposition, with 1 for *true* and 0 for *false*. Certain verbs return Boolean results (for example, tests for equality and membership.)

Storage: Bitwise, 8 items per byte, plus overhead (see below).

[5] Formats are intermixed in the display of *lists*. In tables or higher-rank arrays, all items are currently displayed in the same format.

Integer A number with no fractional part and within the range that can be represented in 32 binary bits: -2^{31} to $2^{31}-1$ (that is, -2147483648 to 2147483647).

Storage: 4 bytes per item, plus overhead.

Floating A number with a fractional part or an integer outside the range representable in *integer* format. Also known as *real numbers*. Floating point numbers may range in magnitude between $\pm 7.2370055773322621\times 10^{75}$. The internal representation uses 14 hexadecimal digits, permitting a maximum precision of about 18 decimal digits.

Storage: 8 bytes per item, plus overhead.

Complex A number with both real and imaginary parts. The parts are written with the letter `j` between them. Each part is represented in Cartesian form, in the same way as a floating point number.

Storage: 16 bytes per item, plus overhead.

In addition to the space required to store its items, there is an additional overhead for system information, including the array's type, rank, and shape, amounting to at least `12+4×` its rank.

Within a numeric array, all items are given the same type of internal representation (Boolean, integer, floating, or complex). For example, the list `1 2` is stored entirely in *integer* representation, even though one of its items (if alone) could be stored as Boolean; similarly, the list `2 2.3` is stored entirely as *floating*, even though one of its items (if alone) could be stored as an integer.

The existence of different internal schemes for representing numbers shows up directly in the space required to store them and may also affect the time required to execute certain verbs. In principle, internal type has *no* effect on the values that a verb can accept as arguments or the values that it returns as result. For example, a verb whose domain is limited to integers can accept any array whose values are numerically integral, regardless of whether the array is stored in the Boolean, integer, floating, or complex internal type.

The interpreter initially assigns the most compact representation that will embrace all the items in an array. Thereafter, upward conversion between types of numeric representation is automatic. If you write

```
x←0 1 1 0
```

the interpreter stores *x* in *Boolean* representation. If you then catenate an integer to *x*, for example by

```
x←x,2
```

the interpreter converts the former value to *integer* representation while appending the integer 2. Similarly, if you then catenate a floating-point number, for example,

```
x←x,1.9
```

the interpreter represents the entire result in *floating* representation.

These conversions are automatic. No declaration of type is needed or allowed. There are no errors resulting from "mismatched" numeric types. The only practical consequences involve allocation of memory, and perhaps execution speed.

However, downward conversion of type, is not performed automatically. For example,

```
x←1 2 3-0 1 3
```

could be represented as Boolean. x will, in fact, be represented as integer.

Character Data

To enter characters from the keyboard, you enclose them in quotes; for example, as `'ABC'` or `'Far away'` or `'3×$6'`.[6] To represent the quote character itself, enter two consecutive quotes. (See the discussion of quotes in Chapter 3, "Grammar".)

Characters usually have no significance as symbols, numbers, or names. (However, the verb *execute* (⍎) instructs SHARP APL to interpret a list of characters as a sentence or series of sentences. See Chapter 6, "Verbs" for the definition of ⍎.)

There is no implicit collating sequence for characters. When you grade (sort) character data, you must provide a reference alphabet to control the order. (See the discussion of ⍋ and ⍒; *grade*, in Chapter 6, "Verbs".

APL Character Set

The SHARP APL interpreter represents each character internally as one 8-bit byte. There are thus 256 possible characters. Not all of them are displayable or enterable from the keyboard. Nor are all of them valid in a sentence to be executed.

6 Strictly speaking, this applies to sentences written during immediate execution or to sentences in the definition of a verb. During input solicited by ⍞, the entire line is treated as characters and hence needs no quotes around it. See the section "Input and Output" later in this chapter.

The character set includes both conventional alphabetic and numeric characters and also characters for the additional symbols used in APL expressions. During output, a few characters are recognized as control characters (for example, *newline*, *line feed* or *backspace*). The bit-patterns that the SHARP APL interpreter uses to represent characters do not match the codes used in the EBCDIC or ASCII encodings. (See also the discussion of characters and keyboards in Chapter 15, "APL Characters".)

The APL system makes use of a larger number of characters than many other computing systems. The manner in which keystrokes from a device are mapped to APL characters, or APL characters are mapped to glyphs in a display, depends both on the particular APL system and the device. These issues are discussed in Chapter 15, "APL Characters".

Boxed Arrays

An item within an array may be a *box*. A box is an item in the same sense that a single number or character is an item: it has no axes, and it occupies a single position in the array's Cartesian framework. An array consisting of six boxes, arranged in two rows and three columns, has shape `2 3`, in the same way as a 2-by-3 array of numbers or characters.

A box may contain within it an array of any rank, shape, or type. A box is thus a convenient way to place objects of arbitrary shape within the simple Cartesian framework of an array. To distinguish it from a noun that contains boxes, a noun whose items are unboxed numbers or characters is said to be *open*.

A box is created by the *box* monad <. The result of < is always an *item* and therefore has rank 0; for example:

```
      ⍴⍴x←<'Now is the time'
0
```

Several ways to form an array of boxes are described in Chapter 6, "Verbs". For example, the *link* verb ⊃ joins a pair of open arrays to form a two-item list of boxes. Because ⊃ boxes its right argument only when the right argument is open, a sentence that links a succession of open arrays produces a list of boxes, thus:

```
      ⍴y←'Now'⊃'is'⊃'the'⊃'time'
4
      y
┌───┐┌──┐┌───┐┌────┐
│Now│└is┘│the│└time┘
└───┘└──┘└───┘└────┘
```

Partitioning

Modifying the *box* verb with the conjunction *cut* produces the derived verb *cut-and-box*. It forms a box from each of the segments in its right argument, using the first cell of the right argument as the delimiter. This *partitions* the right argument into a list of pieces of the argument. For example, to box words delimited by blanks:

```
      z←¯1⍤<⍤0 ' Now is the time'
      y←'Now'⊃'is'⊃'the'⊃'time'
      y≡z
1                                  ⍝ y and z match
      y ≡⍤0 z
1 1 1 1                            ⍝ The items of y and z match

     y⍳<'time'
4                                  ⍝ <'time' is the fourth item
```

Figure 4–2 depicts a 2-by-3-by-4 array of boxes with two of them opened so you can see their contents.

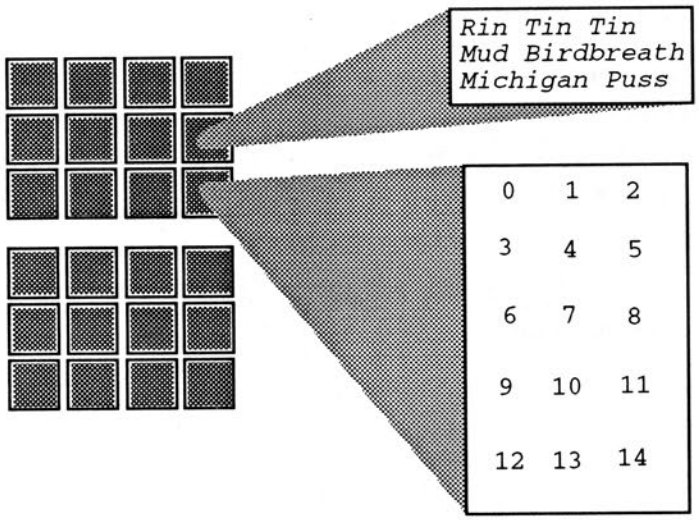


Figure 4–2 : A 2-by-3-by-4 array of boxes.

Display of Boxes

Whenever a sentence produces a result but does not assign a name to it, the interpreter displays it. This applies just as well to an array of boxes as to an array of numbers or characters. Such a default display has the same appearance as the result produced by the *thorn* monad ⍎, described in Chapter 6, "Verbs".

The appearance of a display of an array of boxes is influenced by the value of the system noun ⎕*ps* ("position and spacing").[7] By setting the last two items of ⎕*ps* negative, for example ⎕*ps*←0 0 ¯2 ¯2, you cause the interpreter to show the boundaries of each box. That is particularly useful when you want to make the structure explicit. With the box boundaries thus made visible, the noun *z* (mentioned above) is displayed like this:

```
      ⎕ps←0 0 ¯2 ¯2
      z
|---||--||---||----|
|Now||is||the||time|
|___||__||___||____|
```

Packages

A *package* is a noun that in some respects resembles a *workspace:*[8]

- The items within a package are selected by *name* rather than by position in a framework of axes.
- A package may contain both pronouns and user-defined verbs.
- A package is *not* an array. Hence, verbs that are defined on arrays are *not* defined on packages.

Within a package, the various named objects may be of any type, including not only nouns of any type, but also user-defined verbs. A name in a package may even have *no* referent, so that when the contents of a package are materialized into the workspace, an object having that name is expunged.

A package is useful to provide *overlays*. These permit an application designer to materialize in a workspace – perhaps dynamically, or within a local environment – whichever nouns or verb definitions the application requires, without altering the package from which they are taken. Often the source package resides not in the workspace at all but in a file outside the workspace.

---

7 See Chapter 16, "System Nouns and Verbs" for the definition of ⎕*ps*.

8 See the discussion of *workspace* in Chapter 2, "Tasks, Users, Libraries" and in the first part of Chapter 5, "Naming Nouns and Pronouns".

A package is a data type independent of APL arrays, manipulable only by a set of specialized system verbs (⎕*pack*, ⎕*pdef*, etc.). The contents of a package cannot be displayed directly. A sentence whose result is a package produces as display only the message:

```
**package**
```

To see the definitions of the objects stored in a package, you have to extract the packed items and then display them.

Verbs for manipulating packages are listed below, with square brackets indicating optional arguments. For detailed descriptions, see the entries for the individual verbs in Chapter 16, "System Nouns and Verbs".

[α]	⎕*pack* ω	Form a package.
[α]	⎕*pdef* ω	Define in the workspace objects from a package. (No result.)
α	⎕*pex* ω	Form a package by excluding some objects in a package.
α	⎕*pins* ω	Form a package by merging two packages.
[α]	⎕*plock* ω	Lock the definitions of user-defined verbs in a package.
	⎕*pnames* ω	Return the names of objects in a package.
[α]	⎕*pnc* ω	Return the name-class of objects in a package.
[α]	⎕*ppdef* ω	Like ⎕*pdef*, but protect existing objects by defining only those names that do not conflict with names outside the package.
α	⎕*psel* ω	Form a package by selecting a subset of the objects in a package.
α	⎕*pval* ω	Return the value of a pronoun in a package.

Input and Output

This section deals with the conventions for displaying data at the screen or printing terminal and with entering data from the keyboard. See also the description of the *format* verb ⍕ in Chapter 6, "Verbs", and the description of the system verb ⎕*fmt* in Chapter 16, "System Nouns and Verbs".

Constants

A sentence to be executed may contain characters that represent a number or a list of numbers; similarly, a sentence may contain characters that denote

the characters themselves, as text (rather than names or symbols). Such an entry is a *constant*; it states directly the value of a noun.

The interpreter considers a constant that consists of a single number or character to be an item. An entry that consists of more than one number or character is a list. For example, in the following, `a` is an item and `b` is a list:

```
a←126.2
b←126.3 14 127
```

There is no way to write a constant that has more than one axis. (See "Creating Higher-Rank Arrays," below.)

During entry, blanks separate successive numbers in the same list; extra blanks are not significant.

You may write numbers in different formats within the same list: some in integer format (without a decimal point), others in real format, yet others in exponential format or complex format, as convenient or appropriate for the values you are writing. For example:

```
c←1 .3 0.0125 2.62e¯3 262e2 1.2e4j3
```

A negative number is preceded by the macron, as for example `¯5`. The macron or "overbar" symbol is *not* a verb, but one of the characters used in writing certain numbers, just as the decimal point or the letters `e` or `j` occur in certain numbers. When several numbers are negative, each must have its own macron. By contrast, the symbol – denotes the verb *minus* or *negate* and, like other verbs, applies to an entire array. The expression

```
x←¯12 1.05 4000
```

gives the name *x* to a three-item list; only the first item is negative, whereas

```
x←–12 1.05 4000
```

creates a three-item list all of whose items are negated by the verb (–) that precedes them.

To differentiate characters from numeric data, you enclose a character item (or a character list) in quotes. The beginning of a character constant is marked by a single quote mark. Everything you enter after that (for the balance of the line) is treated as characters until you enter a matching single quote to end the string of characters. Once you have started a quote, *two consecutive quote marks* indicate the quote character itself. (See the discussion of quote in Chapter 3, "Grammar".) Thus, in general, numbers are separated by *blanks* but characters are not. Extra blanks between numbers are not significant. Each of the following creates and names a seven-item list:

```
      x←25 ¯2.4   1e9 2.7e¯24    123456789 70.2j1.4 0
      y←'Thanks!'
      z←'I can''t'
```

Within the zone enclosed by quotes, blank is a character like any other. When you put several consecutive blanks in a character constant, each of them is an item in the same way as any other character in the string.

When the interpreter displays character data, it does not surround it with quotes or double the quote mark, hence:

```
      z
I can't
```

A line containing an odd number of quotes (not counting quotes that occur within a comment) is ambiguous. The interpreter cannot tell which characters are inside and which are outside the quotation. It rejects such a line as a *syntax error*, without executing any of it.

Creating Higher-Rank Arrays

A constant is limited to a *single item* or a *list*. To supply the value of a table (or any array with more than one axis) you have to enter it as a list and then restructure it with a verb such as *reshape* to organize the items into a higher-rank array. Thus:

```
      t←3 4⍴'The fat cat.'
      t
The
fat
cat.
```

Creating the Value of a Box

A boxed constant, like a higher-rank constant, is written as a phrase that includes a verb that forms boxes. In the following example, the verb *link* boxes and joins the constants to form a three-item list of boxes:

```
      h←'Paul' ⊃ 'Hempstead' ⊃ 1933 8 24
Paul Hempstead 1933 8 24       3 1⍴h
Paul
Hempstead
1933 8 24
```

Display of Nouns

Unless the noun produced by evaluating a sentence is assigned a name, the interpreter displays the noun's value. This is called *default display*. It occurs with no need for a "display" verb and requires no instruction regarding format.[9] The display of numbers is conditioned by the visible value[10] of the system nouns ⎕*pp* (printing precision) and ⎕*pw* (page width); the position and spacing of boxed arrays is controlled by the visible value of the system noun ⎕*ps*.

The interpreter displays a list (that is, an array having only one axis) horizontally. It inserts a blank to delimit a number from its neighbor. However, it does not put blanks between the items of a character array. That way, characters can be placed side-by-side to form words, but numbers (which often require several characters to write) do not run together. For example

```
      n←123,45,67.8
```

forms a three-item numeric list. When the interpreter displays the list, it introduces blanks to separate the items:

```
      n
123 45 67.8
```

But it does not insert blanks to separate the items when they are characters:

```
      c←'H','e','l','l','o','!'
      c
Hello!
```

Position and Spacing in an Array of Boxes

During display of an array of boxes, the system noun ⎕*ps* affects the position and spacing of the boxes.[11] In a clear workspace, the value of ⎕*ps* is `¯1 ¯1 0 1`.

When the interpreter displays a table of boxes (or any array of rank greater than 1), it first formats each of the individual boxes separately. It then assembles those separate displays into a grid that retains the overall rectangular shape of the array. To preserve the alignment of the grid, it pads the width of each small box to match the width of the widest box in that column. Similarly, it pads the height of each small box to match the height of the tallest box in that row. The shading in Figure 6–10 illustrates the resulting plaid pattern.

---

9 The default display has the same appearance as the result of the monad ⍕, described in Chapter 6, "Verbs".

10 That is, present in the workspace and not shadowed by the localization of names.

11 See the sections on ⍕ in Chapter 6, "Verbs" and the description of ⎕*ps* in Chapter 16, "System Nouns and Verbs".

Because each item is located in a display cell whose height is the maximum in its row and whose width is the maximum in its column, an item may be assigned to a display cell with more rows or columns than it would need on its own. The first two elements of ⎕*ps* specify how an undersize item should be positioned in the space assigned to it. The values `¯1 ¯1` mean it should go in the top left corner of its box, with any extra space below and to the right.[12]

The last two elements of ⎕*ps* control the way adjacent boxes are separated. The third element sets vertical spacing and the fourth horizontal. The magnitude is the number of additional print positions that should separate adjacent boxes. The default is zero print-positions vertically and one print-position horizontally. Thus:

```
      c⊃c⊃n
Hello! Hello! 123 45 67.8
```

The details of numeric formats, and the arrangement of boxed items in a grid for displaying arrays of rank 2 or higher, are described in the section on ⍕ in Chapter 6, "Verbs".

Separating Successive Columns of a Numeric Array

In displaying a numeric array of rank 1 (a list), the interpreter formats each item individually and separates successive items by one blank.

In displaying a numeric array whose rank is 2 or greater, the interpreter decides for the array as a whole whether to use real or exponential format. When it uses real format, it picks a total width and selects the units position once for the entire array, so that all columns are in a common format.

Separating Successive Planes of Higher-Rank Arrays

A rank-3 array appears as a set of tables. That is, a 3-by-4-by-5 array looks like three 4-by-5 tables, one above the other, with a blank line between the first and second and a blank line between the second and third.

During display, an extra line (or lines) is inserted between successive sub-arrays. *One* blank line is inserted between successive rank-2 sub-arrays, *two* blank lines between successive rank-3 sub-arrays, and so on.

If you examine the result of an expression such as

```
x←⍕Array
```

12 For discussion of these values, see the description of ⍕ in Chapter 6, "Verbs" and the description of ⎕*ps* in Chapter 16, "System Nouns and Verbs".

where `Array` is an open array with more than two axes, you find that `x` has the same rank as `Array`. The display of `x` shows blank lines between its successive tables by virtue of the rank of `x`, not because `x` contains any added *newline* characters.

However, when the system formats an array of boxes, the entire result is represented as a table. In that case, additional rows of blanks are inserted to represent the separation between planes. When `A` is a 2-by-3-by-5 array of characters, and `⎕ps` has its default value,

```
⍕<A
```

represents `A` by a seven-row table whose middle row contains only blanks.

Folding Wide Displays

An array returned by the formatting verbs `⍕` or `⎕fmt` is arranged without regard for the width available to display it. However, when you have output displayed on the screen or terminal printer, SHARP APL takes into account the length of the line available for display. The maximum number of print positions on a line is the visible value of the system noun `⎕pw` (*page width*). The default, `⎕pw` in a clear workspace is site dependent, but a typical value is 132. SHARP APL/PC sets it to 80. You may set it as low as 30 or as high as 250.

The effect of `⎕pw` depends on the rank and type of the array being displayed.

Folding a Character Array

In displaying a character array, the interpreter prints no more than `⎕pw` characters on a line. As soon as it finds it has already put `⎕pw` characters on the current line *and* the next character is something other than *newline*, it forces a new line. It indents the continuation line by six spaces.

A character array is broken solely by the number of characters on the line. No attempt is made to break at a word boundary or other "meaningful" place.

When the array contains an explicit *newline* character (`⎕av[156+⎕io]`), it is obeyed. That is, the next character is displayed at the left margin of the line below, and the length of the current line is counted from there. Thus, a list of characters containing explicit *newlines* appears with the lines folded at the *newline* characters or at `⎕pw`, whichever comes sooner on each line.

Folding a Numeric List

A numeric list is displayed such that each line contains as many numbers as will fit without separating the characters that represent a single number.

Thus, the number of characters on a line is *at most* ⎕*pw*, but may be fewer. Since the number of characters required to represent each number may vary, the number of numbers displayed on a line may vary. So may the number of characters on a line. In rare cases, with values of ⎕*pw* less than 50, a single complex number may be split across two lines.

When a line that will not fit within ⎕*pw* is broken, the next line, and any subsequent lines that are continuations of the original line, are indented by six positions.

Folding a Numeric Table or Higher-Rank Array

A numeric table, or a numeric array of any rank higher than 1, is arranged so that all columns have the same width. SHARP APL figures out the number of print positions that would be required if *all* the items were in a single column. It uses the field width thus calculated for all columns of the array (regardless of the actual widths of numbers in a particular column).

Using the field width just calculated, it starts at the beginning of each row and writes as many fields as will fit within ⎕*pw*. Then it moves to the next line, indents six positions, and prints as many of the remaining fields as will fit within ⎕*pw*−6. It continues in that fashion until the entire row has been printed. Then it prints the next row.

Folding an Array of Boxes

For folding in response to ⎕*pw*, the display of an array of boxes is treated in the same way as character data. That is, each line is broken after ⎕*pw* characters or at a *newline* character (whichever comes first), and the continuation is indented by six positions. SHARP APL makes no attempt to preserve the boundaries of boxes when ⎕*pw* requires folding.

Display of an Empty Array

An empty list appears as a blank line: a line on which nothing is printed. An empty table, or an empty array of any rank greater than 1, produces no display at all (not even a blank line).

This is true not only for a table with zero rows, but also for a table with a positive number of rows but zero columns. Indeed, there is no display at all for any array whose rank is 2 or greater when zero is the length of *any* axis.[13]

13 In this regard, SHARP APL departs from the ISO standard, which simply says that the display of *A* appears the same as ⍕*A*, apparently with the consequence that a 6-by-0 array would display as six empty lines.

Interface Nouns: ⎕ and ⍞

The symbols ⎕ and ⍞ denote the system nouns *quad* and *quote-quad.* They act like nouns shared with the session handler.[14] The action of these two system nouns depends on whether they are *used* or *set.* A noun is said to be *set* when it is immediately to the left of the copula ←, as in:

```
x←foo ⎕←analyze x
```

It is said to be *used* when it occurs anywhere else, as in:

```
x←foo analyze ⎕
```

Assigning a value to ⎕ or ⍞ causes the noun thus assigned to be displayed.

Evaluated Input

In a sentence to be executed, when the symbol ⎕ occurs anywhere *except* immediately to the left of the copula, ← (assuming no redundant blanks), it represents the value of an input expression to be obtained from the session handler.[15]

When, in the course of evaluating a sentence, the interpreter encounters the ⎕ symbol used in this way, it displays the prompt

```
⎕:
```

and awaits one line of entry. It evaluates the next line it receives in the same way that it would evaluate a line entered during immediate execution. That done, it substitutes the resulting value for the ⎕ symbol and resumes execution of the sentence that contained ⎕. For example, suppose you are computing the growth factor of an investment that earns interest for 20 years, compounded monthly. You want a sentence that will prompt you to fill in the annual interest rate but will accept an expression rather than just a number. The following illustrates the use of ⎕ to solicit a sentence whose result is then used in evaluating another sentence.

```
      (1+⎕)*12×Years←20      ⍝Expression containing ⎕
⎕:                           ⍝Prompt for input
      0.08÷12                ⍝One month of 8% annually
4.926802771                  ⍝Result
```

The presence of ⎕ in the phrase `(1+⎕)` causes SHARP APL to issue the prompt ⎕:.

---

14 That is, with the screen display or keyboard input of a T-task or the controlling shared variable of an S-task.

15 That is, from the keyboard of a T-task or by way of the controlling shared variable of an S-task.

In response to the prompt, you enter `0.08÷12`, to represent the monthly interest described by an annual rate of 8%. The interpreter evaluates `0.08÷12` and substitutes the result `0.006666666667` for ⍞ so that it can evaluate `1+⍞`. That done, it raises `1.006666667` to the power `12×20` and prints the result, indicating that the value after 20 years is nearly five times the initial amount.

Input in Response to ⍞

In response to the ⍞: prompt, you enter one *line* of input. It may contain any executable sentence,[16] invoking any primitive verb or any user-defined verb whose name is visible. When evaluation of that sentence is complete, the interpreter passes its value back to the expression that contained the ⍞. In the example, the result is used as the right argument to +, so the result could be anything in the domain of +.

When, in response to the ⍞: prompt, you enter a line containing several sentences separated by diamonds, the interpreter evaluates all of them, but the value substituted for ⍞ is the value returned by the *last* sentence to be executed.

When, in response to the ⍞: prompt, you do something that produces no sentence to evaluate (for example, you simply press *Enter* without writing a sentence, or execute a system command), the interpreter repeats the ⍞: prompt for evaluated input.

It is acceptable to enter a system command during ⍞-input. The interpreter carries out the command. However, since in this case the command does not satisfy the request for an input sentence, the interpreter reissues the ⍞: prompt. You could exploit this to copy into the workspace something that was not initially present and then use the object thus copied in your response to the prompt. For example:

```
Supply data for analysis:
⍞:
      datapoints
value error
      datapoints
      ^
⍞:
      )copy savedwork datapoints
saved 1988-02-27 23:11:15
⍞:
      datapoints
...
```

16 Or several sentences separated by diamonds.

When, in response to the □: prompt, you enter → alone (the naked right arrow), the interpreter abandons both the execution of □ *and* the execution of the sentence that contained it, as well as anything else pending, back to the previous immediate-execution entry from the keyboard. (See Chapter 9, "Control of Execution".) A gentler method of halting execution is to enter an *input interrupt* (see below). Execution of `)load` also abandons execution of the sentence, as well as the entire application.

Character Input

In a sentence to be executed, when the symbol ⍞ occurs anywhere *except* immediately to the left of the copula, assuming no redundant blanks, it represents a character list to be obtained from the keyboard. The interpreter displays no special prompt (except as may be produced by ⍞ output, described below). The interpreter awaits the entry of one line from the keyboard. It treats that line as characters (even if what you then enter looks like a sentence or a system command). It substitutes the resulting characters for the ⍞ symbol and resumes execution of the sentence that contained ⍞. For example:

```
'Enter data:' ⋄ (⍎⍴c←⍞),' received.'
```

```
Enter data                    ⍝Display of the first sentence
abcdefghijklmnopqrstuvwxyz    ⍝Your entry
26 received.                  ⍝Display from ⍎⍴c
```

Escape from ⍞-Input and □-Input

While the interpreter is waiting for one line of character or evaluated input, you can signal an *input interrupt* and thereby *escape* from the request for input.

The manner in which you signal an *input interrupt* depends on the session handler and the device. From some devices, it is the five-character sequence *o*-backspace-*u*-backspace-*t* sent from the keyboard. To an S-task, there is a special code that means "this is an input interrupt" (see Chapter 12, "S-tasks").

Sending an input interrupt at any point during your response to a request for ⍞-input – that is, at a keyboard any time *before* you press *Enter* or *Return* – causes the interpreter to abandon execution of the sentence that contains the ⍞ or □ and to halt execution of the program that contains that sentence. Any characters you may have already entered on the same line are lost. However, if `□trap` has been set to respond to the input interrupt event, the trap may make it difficult or impossible to escape. See Chapter 10, "Event Handling".

Explicit Output from ⎕ and ⍞

When the system name ⎕ or ⍞ occurs immediately to the left of the copula ←, the interpreter displays the value of the noun to the right of the arrow. A display produced this way is identical to a display produced by default output or by the monad ⍕, with one important exception: output produced by ⍞← is not followed by a *newline*, whereas the interpreter automatically supplies a *newline* following default output or output elicited by ⎕←.

A series of sentences produced by ⎕ appear on successive lines, whereas a series of sentences produced by ⍞ output may accumulate on the same line. This allows a line of output to be built up from the results of several expressions. For example, a program might contain the following sequence:[17]

```
[ ]   ...
[ ]   year←years[i]
[ ]   dist←recordmiles[i]
[ ]   ⍞←year
[ ]   ⍞←''''s record was '
[ ]   ⍞←dist×1.6
[ ]   ⎕←' km.'
[ ]   ⍞←'Today''s distance: '
[ ]   ⍞←todaymiles×1.6
[ ]   ⎕←' km.'
[ ]   ...
```

When that fragment of the program is executed (with appropriate values for the various names), the separate pieces of ⍞ output fit together thus:

```
1982's record was 89.76 km.
Today's distance: 90.11 km.
```

However, it is probably easier and faster to build the entire output array, then emit it with one output request, than to use the piecemeal approach shown here.

Folding of ⍞ Output

Output produced by ⍞ is subject to ⎕*pw* in the same way as other character output. That is, SHARP APL ends each line when it has displayed ⎕*pw* characters and the next character is not *newline*, or at a *newline* character (whichever comes first). On each continuation line, it inserts six blanks at the beginning of the line.

17 These have to be in a program. If you were to enter these lines from the keyboard for immediate execution, the interpreter's prompt for your next entry would obscure the effect.

Input on the Same Line as the Character Prompt

You may want to display a prompt and accept input on the same line. You can do that by placing ⍞ output before a request for ⍞-input. For example:[18]

```
      x←⍞⊣⍞←'Name: '
Name: John
```

In the example above, the noun *x* receives both the characters *John* entered from the keyboard *and also characters representing as much of the prompt as appeared on the same line with the reply.*

Where the prompt takes more than one line (for example, because the prompt is a table, or contains a *newline* character), only the part of the prompt that occurs on the *same* line as the input is represented in the result.

When the interpreter has displayed the prompt and is ready to receive your reply, the cursor is at the next position to the right. Using _ to show where the cursor is, it looks like this:

```
Name: _
```

If you simply key the four letters *J o h n* without first moving the cursor somewhere else, the result contains ten characters. The first six are the characters from the prompt, and the last four are the letters you entered: *John*.

However, if you move the cursor leftward into the prompt zone and there enter some other characters (overwriting the characters used in the prompt), the characters you substitute are reproduced in the result.

Discarding the Prompt

When you have no interest in the prompt or in entries placed in the prompt zone, you may simply discard them from the result. Phrases such as the following are common:

```
prompt←'Supply name: '
name←(⍴prompt)↓⍞⊣⍞←prompt
```

Alternatively, you can exploit a quirk of ⎕*arbout* to eliminate the prompt from the result:

```
prompt←'Supply name: '
⍞←prompt ⋄ ⎕arbout '' ⋄ name←⍞
```

[18] The example could also be written with prompt and reply on separate lines. However, in writing a program it is prudent to keep prompt and reply on the same line so that they remain together even if the program is halted and restarted. It is common practice to use a utility user-defined verb, such as *ask*, for this purpose.

Interposing any use of `⎕arbout` between the input and output uses of ⍞ *discards* the output from the ⍞-input buffer and prevents the prompt zone from being treated as part of the input line. (The verb `⎕arbout` is described in Chapter 16, "System Nouns and Verbs".)

5 *Naming Nouns and Verbs*

This chapter discusses how names are used and the means by which nouns and verbs acquire names.

←

Copula

The left arrow ← assigns a name to something. For example

```
x←analyze data
```

gives the name *x* to the noun that results from evaluating the expression

```
analyze data
```

The symbol ← plays in APL the same role that the copula "is" plays in English. An expression such as

```
area←6×8
```

is often read as "Area is six times eight." Because the arrow ← *assigns* a name to an object,[1] it is often called *assignment.*

Once a noun has been named, further reference to the name is a reference to the noun's value; for example:

```
      area←6×8
      area
48
      area*0.5
6.92820323
```

Workspace: The Universe of Names

In order to evaluate a sentence that contains names, the interpreter requires some means of keeping track of the names that have been defined and what each name refers to. The mapping from names to their referents is provided

[1] In SHARP APL, the copula can only be used to assign a name to a noun (although "A Dictionary of APL" also permits ← to assign a name to a verb).

by the *symbol table*. The set of all names maintained by a symbol table, along with their meanings, is called a *workspace*.[2]

A workspace is not only the domain in which a common set of names is maintained by a symbol table, but also a block of physical storage. The area of the computer in which computation takes place is called the *active workspace*. Each APL task has one (and only one) active workspace. The system command `)save` sets aside a copy of the active workspace's symbol table and all the objects referred to in it and retains it in inactive permanent storage. A workspace is thus both the conceptual universe in which names have meaning and a unit of storage. By *saving* a workspace, you set aside a copy of it. By subsequently *loading* a saved workspace, you restore in the active workspace all the names and all their referents as they were at the time the workspace was saved.

Localization of Names

A user-defined verb[3] may declare certain names to be *local* to itself. This means that the verb (or any verb it invokes) may make its own use of the names local to it, independent of any use those names may have outside the verb. Localization has these effects:

- It assures the verb's author that internal names needed by the verb will be available to it.
- It protects names in the workspace from possible conflicting uses within the verb's definition.
- It hides more global use of those names, if any, since from within a verb (or any verb it invokes), only the most local use of a name is visible.
- It cleans up after execution of a verb. When execution of a verb is complete, that ends its local use of names; the named objects it created vanish, and the interpreter frees any storage it had devoted to them.

A name is made local to a verb by appearing in the verb's *header*. The names used for the *arguments* and *result* of a user-defined verb are automatically local to it. So are any *line labels* used in the verb's definition. (Localization of names is discussed further in Chapter 8, "Verb Formation" and Chapter 9, "Control of Execution".)

---

2 In many other computer systems, the names used by the various programs are harmonized by a process called *linking*; the program that links separate components so that they can be used together is called a *link editor*. In APL systems, the consistent use of names is provided directly by storing objects in a workspace whose common symbol table gives the benefits of linkage without any specific process of link editing.

3 See Chapter 3, "Grammar" and Chapter 8, "Verb Formation".

When none of the currently-active verbs has localized a particular name, that name is *global* to the workspace. But when a name has been localized in a defined verb currently being executed, that local use is understood.

When a name is *not* local to the current verb, but *is* local to a pendent verb (a verb whose execution has been started but not completed), the use local to the pendent verb is understood.

A declaration of local use prevents seeing, changing, or creating a use of that name at a more global level. (It is the declaration that counts; it does not matter whether the localized name has actually been used.) Local use blocks the view of more global use; the most local use of a name is sometimes called the *visible* use of the name; a more global use is called a *shadowed* occurrence of the name.

Reassignment and Name Conflicts

When you use the copula to name something, the name to the left of the arrow must be *free for use*; that is, it must have no visible use as a verb or as a label.

- When there has been no prior use, the copula assigns the name to the left of the arrow to the noun to the right of the arrow.
- When the name is already in use and refers to a noun, the value to the right of the ← *replaces* the former value. (That is why the name of a noun is also called a *variable*: its value can be changed at will.)

Names for User-Defined Objects

Both nouns and verbs may be given arbitrary names. As mentioned earlier, a noun is assigned the name that appears to the left of the ←. Verbs may also be named, but by a different mechanism (described in Chapter 8, "Verb Formation").

A *name* must start with a letter of the alphabet; subsequent characters in a name may be letters or numerals. The symbols Δ and ∆ are both permitted in names in the same way as letters. A name may not contain more than 77 characters.

The symbols ⍺ (alpha) and ⍵ (omega) may also be used as names, but only when used *alone*, and not as part of a longer name. There are no rules regarding the uses to which you put the names ⍺ and ⍵, but by convention ⍺ is used as a generic name for a verb's left argument, ⍵ for a verb's right argument.

Since the arbitrary name assigned to a noun is a shorthand reference to the noun, the name serves as a *pronoun*. As with pronouns in natural language, separate processes may assign different names to the same noun; see the discussion of *shared variables* in Chapter 11, "Shared Variables".

APL systems have little that corresponds to the *declarations* of other languages.[4] You do not have to declare the type of noun to which a name will refer, or its size. When you use a name, the interpreter simply notes the use made of it. The value to which a pronoun refers may be reassigned at any time. Reassignment may change its type, rank, or shape, as well as its value.

The only form of declaration is *localization* of a name to the definition of a verb. Names that are *local* to the definition of a verb must be declared as part of the verb's definition.

Indexed Assignment

The name to the left of the ← may be modified by index brackets, in an expression of the form:

```
x[i] ← b
```

The effect is to produce a new version of x in which the items at position i are taken from b, while the others are retained unchanged from x.[5] Indexed assignment is also discussed as part of the description of indexed selection in Chapter 6, "Verbs".

Indexed assignment requires that:

- The array x must already exist.
- The expression within brackets must be appropriate to the rank of x (that is, must have the right number of semicolons: `¯1+⍴⍴x`).
- The values of the indexes `[i]` must be valid, given the index origin and the shape of x.
- The shape of b must match the shape implied by the index expressions within the brackets. b may be a single item, in which case it is inserted at *all* the locations indicated.[6]
- x and b must be of the same type (numeric, character, or box).

---

4 For example, in languages such as C or Fortran, when you write a program you have to tell the compiler in advance what names you plan to use and the type and rank of each. They also require you to state (explicitly or implicitly) the maximum shape that each object will have, so that the compiler can set aside space for it.

5 In "A Dictionary of APL", this is achieved by `b i⍤{}x`, which has the advantage that the result is the entire modified array x, whereas the formal result of `x[i]←b` is b.

6 The interpreter also tolerates a mismatch between the ranks of b and x, but only when the additional axes are of length 1.

The Result of Assignment

When a verb occurs to the left of an assignment, the verb's argument is the result of the assignment. The result of assignment is the noun to the right of the ←. For example, in the sentence

```
3×x←analyze data
```

the right argument of × is the result of `analyze data`. Such a sentence can be read, "Three times *x*, which is `analyze data`."

This is equally true for indexed assignment. In the sentence

```
3×x[i]←a
```

the right argument of × is the value of `a` (and *not* the value of *x*).

Multiple Assignment

A sentence may contain several assignments in a row. For example,

```
x←y←z[i]←a
```

assigns the value of `a` to the name `x` and also to the name `y`. It also replaces the value of the sub-array `z[i]`. This does not make `x` and `y` synonyms; in particular, if you subsequently change the value of `x`, the value of `y` does not change.

Sentences Producing No Display

The interpreter displays the result of every sentence *except* a branch, an assignment, or one containing a root verb that does not return a result.[7]

An assignment sentence is a sentence that has ← at the root. In an assignment sentence, the name to the left of the copula is assigned to the noun to the right, but there is no display.

Placing ⊢ or ⍞← to the left of the principal assignment permits the same sentence to assign a name and display its value:

```
      ⊢x←(⍳5)∘.=⍳5
1 0 0 0 0
0 1 0 0 0
0 0 1 0 0
0 0 0 1 0
0 0 0 0 1
```

[7] Types of sentences and a sentence's root verb are discussed in Chapter 3, "Grammar". Branch is discussed in Chapter 9, "Control of Execution".

Other Mechanisms that Produce Named Objects

As discussed in the first part of this chapter, the copula ← assigns the name at its left to the object at its right. Other mechanisms that may also produce a named object are the following:

`⎕fx` `3 ⎕fd`	As a side-effect, these system verbs create a verb from a right argument that includes both the verb's definition and its name. The verb thus created becomes the *local* use of the name. (See Chapter 8, "Verb Formation" and Chapter 16, "System Nouns and Verbs".)
∇-editor	The ∇-editor (invoked by the symbol ∇) creates a verb (or modifies an existing verb). The name of the verb appears in the top (header) line of the definition in the same manner as for `⎕fx` or `⎕cr`. The verb thus created becomes the *global* use of the name. (See Chapter 8, "Verb Formation".)
`)copy` `)pcopy`	These system commands copy into the active workspace some or all of the names in the specified workspace and the objects to which the names refer. The objects thus defined become the *global* uses of the names. (See Chapter 17, "System Commands".)
`⎕pdef` `⎕ppdef`	This system verb materializes in the workspace nouns or verbs stored in a package. The objects thus materialized become the *local* uses of the names. (See Chapter 16, "System Nouns and Verbs".)

6 *Verbs*

This chapter describes each of the *primitive verbs* of SHARP APL, together with a few examples to illustrate its use. It also describes the indexing primitives. It includes every verb denoted by an APL *symbol*. However, this chapter does not include the *system verbs*, which are denoted by a *distinguished name* beginning with ⎕ (such as ⎕*read*). System verbs are described in Chapter 16, "System Nouns and Verbs".

There is no inherent order for the symbols; they are arranged rather arbitrarily. The familiar + − × ÷ come first. After that, symbols that are similar in appearance are grouped together; so that ⌸ comes next to ÷; ⋆ ⊛ ○ are adjacent; and so on.

Names are given with the descriptions. Note, however, that a symbol may be associated with several names. (That is why they are not in alphabetical order of their names.) The descriptions may distinguish between:

- a name for the verb's *symbol*
- a name for the *dyad*; that is, the case when the verb is used with an argument on either side
- a name for the *monad*; that is, the case when the verb is used with a single argument.

The variety of names reflect the wide variety of names in common use. "A Dictionary of APL" proposes short names borrowed from a variety of English contexts. In some cases, those are included here, along with alternatives used in programming or mathematics.

In chemistry, the terms *monad* and *dyad* refer to objects having valence 1 and valence 2 respectively. That is how they are used here. Thus, the dyad ÷ is *divide* (as in *a*÷*b*), while the monad ÷ is *reciprocal* (as in ÷*x*).

In the descriptions, the following symbols appear:

- ⍺ The verb's left argument.
- ⍵ The verb's right argument.

Default Argument Ranks

A verb's *rank* is the number of axes in each argument cell.

Whenever you apply a verb to an argument of rank greater than the rank for which the verb is defined, the "extra" axes are treated as *frame* axes. The verb is applied independently to each cell within that frame. The rank conjunction ⍤ (see Chapter 7, "Adverbs and Conjunctions") can impose on any verb the specific argument rank specified by the noun to the right of ⍤.

In principle, every verb has a *default argument rank*. When you use the verb *without* explicitly saying what argument rank you want, SHARP APL assumes you mean the default argument rank. The default argument ranks are shown with the description of each verb. They are also summarized in Figure 6–1.

Argument rank is shown here (and in the argument of the rank conjunction ⍤) as a three-item list of integers. The items are:

- *Monadic rank.* Rank of an argument cell when the verb is used with one argument.
- *Left dyadic rank.* Rank of a left argument cell when the verb is used with two arguments.
- *Right dyadic rank.* Rank of a right argument cell when the verb is used with two arguments.

Agreement

The arguments of a dyad must have frames whose shapes *agree*. The two frames agree when:

- they have the same shape; that is, the same number of axes and the same length along each axis

  or

- one frame is empty; that is, has *no* axes.

When a frame is empty, the argument consists of a single cell. When one argument has an empty frame but the other does not, the frame that is not empty establishes the frame for the result. The single cell from the empty frame is paired with *each* of the cells in the other argument.

Scalar Verbs

In mathematics, a *scalar* is an object that has *no* axes (in this manual, usually called an *item*). A *scalar verb* is one that is rank-0 and returns a rank-0 result.

Because a scalar verb is defined without reference to axes, when you apply it to an array argument, *all* the argument's axes are *frame* axes. Therefore, the verb applies independently to each of the items throughout the frame.

When a scalar verb is used dyadically, the usual rules apply: either the two frames must have the same shape or one of them must contain only a single item, to be paired with every cell in the other argument. For example, adding the numbers contained in frames of length 3 produces three independent additions of the corresponding cells:

```
      1 2 3 + 12 5 20
13 7 23
```

That is because `1+12` is `13`, `2+5` is `7`, and `3+20` is `23`.

Extending the Empty Frame

When one frame is empty, the single cell corresponding to that empty frame is paired with each cell in the other argument; the empty frame is said to be *extended* to match the frame of the other argument. The result frame has the same shape as the non-empty argument frame. This extension is illustrated in the following examples for addition:

```
      111 + 4.9 ¯0.012 34
115.9 110.988 145

      4.9 ¯0.012 34 + 111
115.9 110.988 145
```

Unbounded Rank

A verb may have *unbounded* rank (sometimes called *infinite* rank). When a verb has unbounded rank, no matter how many axes the argument has, they are all treated as part of a single cell. In the descriptions that follow, unbounded default rank is denoted by the ∞ symbol.

Unspecified Rank

For some verbs, no default argument rank has yet been specified. (The fact that some argument ranks remain unspecified is not a feature of the language, but a limitation of the current implementation.) You can still use the rank conjunction ⍤ to specify a verb's argument ranks, or you can supply an argument of the usual rank. But if you apply such a verb to an argument that has more axes than the "usual" case, SHARP APL does

not know how to partition the axes into frame and cell. It reports a `rank error`. The verbs in Figure 6–1 whose entries contain an asterisk (*) are those for which a default rank has not yet been specified.

Reduction and Identity Elements

The class of derived verbs called *reductions* (see Chapter 7, "Adverbs and Conjunctions") applies the same verb between each of the major cells of its argument. For example, if an array X consists of three major cells X_1, X_2, and X_3, then the f reduction of X, written $f{\neq}X$, is computed by:

$$X_1 \; f \; X_2 \; f \; X_3$$

A reduction applies to as many cells as the argument has; the expression $+{\neq}X$ computes the sum of two cells when X has two cells, the sum of three cells when X has three cells, and so on.

The verb used in a reduction is a dyad. Reduction is valid even when the argument consists of only one cell, or zero cells. What does it mean to apply a dyad to one cell, or to no cells? Ordinarily, "the sum of one thing" might seem nonsensical, and "the sum of no things" even more so. A useful generalization is obtained by the following rules:

- When ω contains only one major cell, the result of any reduction is ω.
- When ω contains no major cells, the result of any reduction of ω is the verb's *identity element.*

The reason for this becomes apparent when you consider how a final result is constructed from a set of partial results. It is often useful to partition a task; that is, to perform the task for part of the data, then another part, and so on. A final step consolidates the results from the parts. Partitioning is possible when a simple rule relates the result obtained from the various partitions to the result for the array as a whole. For example, if an array has five cells, you could sum the first two cells, then the last three, and finally sum those results to get the overall sum.

(Sum of all five) = (Sum of first two) + (Sum of last three)

(Sum of all five) = (Sum of first three) + (Sum of last two)

... and so on.

More generally, you should be able to split the five cells anywhere; summing the separate partitions and then summing the results should still give you the same result in the end. This principle helps define what reduction should mean when it is applied to one cell, or no cells. Generalizing from the preceding equation, we get the following:

Verb	Monad Rank	Dyad L Rank	Dyad R Rank
+	0	0	0
−	0	0	0
×	0	0	0
÷	0	0	0
⌹	2	∞	2
*	0	0	0
⍟	0	0	0
○	0	0	0
⌈	0	0	0
⌊	0	0	0
|	0	0	0
!	0	0	0
?	0	*	*
∪	∞	∞	∞
<	∞	0	0
>	0	0	0
≤	Undefined	0	0
≥	Undefined	0	0
=	Undefined	0	0
≠	∞	0	0
≡	Undefined	∞	∞
~	0	Undefined	Undefined
⌶	Undefined	Undefined	Undefined

Verb	Monad Rank	Dyad L Rank	Dyad R Rank
∧	Undefined	0	0
∨	Undefined	0	0
⍲	Undefined	0	0
⍱	Undefined	0	0
⊣	∞	∞	∞
⊢	∞	∞	∞
⊤	Undefined	∞	∞
⊥	Undefined	∞	∞
⍕	∞	*	*
⍎	*	Undefined	Undefined
⍪	∞	∞	∞
,	∞	∞	∞
⍴	*	*	*
⌽	*	*	*
⊖	∞	*	*
↑	Undefined	*	*
↓	∞	*	*
⍋	∞	∞	∞
⍒	∞	∞	∞
⍳	∞	*	*
∈	Undefined	0	∞
⍷	Undefined	∞	∞
⍸	*	*	*

Undefined: No meaning established for this verb form.

*: This verb form has meaning, but no default argument rank has been specified for it.

Figure 6–1 : Default argument ranks of primitive verbs.

(Sum of all five) = (Sum of first one) + (Sum of last four)

(Sum of all five) = (Sum of first none) + (Sum of last five)

It becomes evident that "the sum of the first one cell" should be the first cell itself. The sum of no cells should be something that makes no difference, since "sum of the last five" already gives the result "sum of all five."

For a particular dyad, the identity element is the value which, when it is one of the verb's arguments, guarantees that the verb's result is equal to the other argument. `0` is the identity element for addition because `0+`*anything* gives you *anything*, unchanged. Similarly, `1` is the identity element for multiplication, because `1×`*anything* gives you *anything*, unchanged.

For some verbs, there is no identity element. For others, there is a right identity but not a left. For example, *anything*`÷1` gives you *anything*, but that is not true for `1÷`*anything*.

In the individual descriptions that follow, the value of a verb's identity element is noted where it has one. The identity elements are summarized in Figure 6–2. (See also the discussion of the ⌿ and / adverbs in Chapter 7, "Adverbs and Conjunctions".)

Dyad	Identity Element
+	0
−	0
×	1
÷	1
=	1
≠	0
<	0
>	0
≥	1
≤	1
*	1

Dyad	Identity Element
⊛	None
|	0
!	1
○	None
∧	1
∨	0
⍲	None
⍱	None
⌈	Smallest representable number ¯7.2370055773322621e75
⌊	Largest representable number 7.2370055773322621e75

Figure 6–2 : Identity elements for scalar dyads.

Result Rank and Shape

For many verbs, a result cell has the same rank and shape as its argument cells. Wherever the rank or shape of a result cell is different from the rank or shape of the argument cells, the description notes that fact.

Verb Definitions

+ − × ÷

Plus, Minus, Times, Divide

These four dyads are familiar from elementary arithmetic. The monads are also defined, and given names as follows:

	Monad	***Dyad***	*Identity Element*	*Argument Rank*
+	Conjugate	Plus	0	0 0 0
−	Minus; Negate	Minus	0	0 0 0
×	Signum	Times	1	0 0 0
÷	Reciprocal	Divide	1	0 0 0

Dyads + − × ÷

The dyadic meanings are those of arithmetic.

For complex arguments, the usual rules apply. That is, the sum (or difference) of a pair of complex numbers is the sum (or difference) of their real and imaginary parts considered separately:

```
      2j4 + 1.5j2.1
3.5j6.1
```

The fact that the sum of two complex numbers is represented by the sum of their real and complex coordinates taken separately is something you can exploit in graphics applications. Using complex numbers to describe a set of points in a plane, you can compute a linear translation of a set of points simply by using + to add the amount of change in the two axes.

Cartesian Representation of Complex Product

The product of a pair of complex numbers can be stated in terms of operations on their Cartesian representations. Suppose a verb `Ct` decomposes a complex number into its real and complex parts, and the inverse `Cx` reconstructs a complex value from such a representation. Then the product of two complex numbers ⍺ and ⍵ is given by the following definition of `Times`: [1]

[1] These definitions are shown in a compact style called *direct definition.* Each verb's definition takes a single line. The verb's name is to the left of the colon. Its left argument is represented by ⍺ and its right argument by ⍵. The result of the sentence is the verb's result. For more detailed treatment of this form of definition, see "A Dictionary of APL," Table 3, p. 39.

```
Times: Cx (-/α×ω) ,⍤0 (+/α×⌽ω) ⊣ α←Ct α ⊣ ω←Ct ω
   Ct: ⍉⍪  9  11 ∘.○1 0 ω
   Cx: +/ ¯9 ¯11 ∘.○1   ω
```

The complex value is the real part plus i times the complex part, where `i←¯1*0.5`. The expression `+/¯9 ¯11○` achieves that because `¯9○ω` is defined to be ω, while `¯11○ω` is defined to be i times ω.)

Polar Representation of Complex Product

The product can also be computed when each complex number is represented by its magnitude and phase angle. In that case, the product of two of them has:

- magnitude equal to the product of their magnitudes
- phase equal to the sum of their phases.

Suppose that a verb *MP* decomposes a complex number into its magnitude and phase angle, and the inverse *CMP* reconstructs a complex value from such a representation. Then the product of two complex numbers α and ω is given by the definition of *Times*:

```
Times: CMP (1 0/α×ω), 0 1/α+ω  ⊣ α←MP α ⊣ ω←MP ω
   MP: ⍉⍪  10  12 ∘.○1 0 ω
  CMP: ×/ ¯10 ¯12 ∘.○1   ω
```

Rotation of a set of points in the complex plane through an angle of θ radians is equivalent to multiplying by a number having magnitude 1 and phase θ; you can exploit this identity in graphics applications that use complex numbers to describe a set of points in a plane.

Division by Zero

As in arithmetic, dividing a non-zero quantity by zero is undefined and the interpreter rejects it as a *domain error*. However, to preserve some convenient identities, SHARP APL defines `0÷0` to be `1`.[2]

Monad +

`+ω` gives the *conjugate* of ω, also called the *complex conjugate*. Formally:

```
+ω  ⟷  (|ω×ω)÷ω      When ω≠0
+ω  ⟷  0             When ω=0
```

[2] "A Dictionary of APL" follows McDonnell's argument that `0÷0` should be `0` ("Zero Divided by Zero," *ACM Quote-Quad, APL76*, pp. 295-302). However, the convention that `0÷0` is `1` was adopted by all early implementations of APL, and is included in the ISO standard for APL.

The conjugate is the value you get when you reflect the argument around the real axis. The real part is unchanged, but the sign of the imaginary part is reversed. For example:

```
      +9 7j3 ¯7j2 8j¯3 ¯2j¯3
9 7j¯3 ¯7j¯2 8j3 ¯2j3
```

Thus, $+\omega$ differs from ω only when ω is complex.

Monad −

The result has the same magnitude as the argument, but opposite sign. For a complex argument, the result reverses the signs of both the real and the imaginary part.

Monad ×

$\times\omega$ gives the *trend* of ω. For real arguments, that is equivalent to the *signum*; the result is:

`1`	where ω is positive;
`0`	where ω is `0`;
`¯1`	where ω is negative.

Formally, the trend is defined by:

$\times\omega$	$\Longleftrightarrow$	$\omega\div\lvert\omega$	When $\omega\neq 0$
$\times\omega$	$\Longleftrightarrow$	`0`	When $\omega=0$

For a complex argument, the result is the point at which a ray from the origin to ω intersects the unit circle.

The trend is *not* affected by `⎕ct`, comparison tolerance. See the definition of equal (=) for a definition of tolerant comparison.

Monad ÷

$\div\omega$ is defined as $1\div\omega$, that is, as *reciprocal*. As in arithmetic, reciprocal is undefined when the argument is `0`, so you get a *`domain error`*.

⌹ Matrix Inverse; Matrix Divide

	Monad	***Dyad***	*Identity Element*	*Argument Rank*
⌹	Matrix inverse	Matrix division	[none]	2 ∞ 2

The ⌹ verb *domino* is a generalization of the mathematical function *matrix inverse*. Mathematicians have no special symbol for matrix inverse, but indicate it by writing M^{-1}. Matrix inverse is analogous to the reciprocal because, just as (for a non-zero number `n`)

```
1 ⟺ (÷n) × n
```

for a non-singular matrix `M`

```
I ⟺ (⌹M)+.×M
```

where `I` is the *identity matrix* (a square matrix having the same number of columns as `M`, with ones along the diagonal and zeros elsewhere) and `+.×` is the matrix dot-product, an analog of multiplication for matrices.[3]

Just as not all numbers have a reciprocal (zero has no reciprocal), not all matrices have inverses. Matrices with no inverse are called *singular matrices*.

Continuing the analogy with division of numbers, matrix division satisfies the identity (for two appropriately shaped matrices `A` and `M`, and `M` not singular).

```
a ⟺ (a÷n) × n          (Numbers)
A ⟺ (A⌹M)+.×M          (Matrices)
```

If you attempt to use a singular matrix as the right argument to ⌹, SHARP APL rejects the expression as a `domain error`.

Monad ⌹

The inverse of a non-singular matrix `M` is found by ⌹`M`. The monadic use ⌹`M` is equivalent to `I`⌹`M`. The shape of each result cell is the reverse of the shape of an argument cell.

Dyad ⌹

In general, the matrix quotient α⌹ω is linked to the inverse of a matrix by the identity:

[3] See the discussion of *inner product* in the section on the *dot* (`.`) conjunction in Chapter 7, "Adverbs and Conjunctions".

```
α⌹ω  ⟺  (⌹ω)+.×α
```

When ω is square (that is, has as many rows as columns), the solution is exact. When ω has more rows than columns, the result is the least-squares approximation that minimizes the sums (down the columns) of the squares of the difference between α and $\omega+.\times\alpha\,⌹\,\omega$.

For a matrix *M* and column-vector (that is, a one-column matrix) *c* having the same number of rows as *M*,

```
c⌹M
```

is the solution to a set of linear equations:

c The *constant terms*.

M A matrix containing the *coefficients* for each of the unknowns.

M has a column for each unknown, and a row for each equation. Since you must have at least as many equations as unknowns, *M* must have at least as many rows as it has columns. When *M* is square, and has the same number of rows as *c*, then:

```
c ≡ M +.× c⌹M
```

When *M* has more rows than columns (and there is a row of *M* for each major cell of *c*) then *c*⌹*M* is the least-squares approximation, defined to minimize each of the column-sums

```
+⌿d×+d ⊣ d←c−M+.×c⌹M
```

where *d* is the difference between *c* and *M*+.×*c*⌹*M*. When the solutions are real, *d*×+*d* is equivalent to *d**2.

SHARP APL treats a rank-1 left argument as if it were a one-column matrix. (See the discussion of "Frame and Cells with Dyad ⌹", below.)

Example: Multiple linear regression requires a list of dependent observations in the column-vector *c* and a matrix of independent or *predictor* observations *p*. The matrix *p* has a column for each of the independent variables and a row for each row of *c*. (Since SHARP APL accepts a list in the place of a column vector, that may be an item of *c* for each row of *p*.) The coefficients of the best-fitting linear combination of the independent variables are obtained from *c*⌹*p*.

Example: Finding a best-fitting polynomial is essentially the same procedure but with a single independent variable. The columns consist of the list of observations of that variable, each raised to the successive powers by an expression. Hence, to find the coefficients of the best-fitting polynomial of

degree *d* to approximate the observations ⍺ from a *predictor* variable *p*, the expression is: [4]

```
⍺ ⌹ p∘.*0,⍳d
```

When ⍺ has multiple columns, the result is a matrix containing in each of its columns an independent solution for each of the columns of ⍺.

Frame and Cells with Dyad ⌹

The left argument of matrix division has *unbounded* rank. That is, the left argument is treated as a single cell, regardless of the number of axes. The right argument rank is 2. Thus, when the right argument has more than two axes, any additional axes constitute the frame in which the rank-2 cells are embedded. Each of the ⍵-cells is paired with the single cell formed by the entire ⍺.

The length of the first axis of ⍺ must match the length of the first axis of each ⍵-cell. Those matching axes disappear from the result, as they do in reductions. The remaining axes of ⍺ after the first – call them ⍺'s *trailing* axes – contain the constant terms for independent sets of equations to be solved, each set in a column of ⍺. There is a result vector corresponding to each of ⍺'s column vectors.

The various columns of ⍺ do not interact. Suppose each of the ⍵-cells has *r* rows. Then the first axis of ⍺ must have length *r* to match. Each of the columns of ⍺, however arranged, contains a separate set of constants for which a separate solution will be found. For example, if there are six such columns, it makes no difference to the calculation whether they are arranged so that ⍺ has shape `(r, 6)`, or `(r, 6 1)`, or `(r, 2 3)`, or `(r, 3 2)`, and so on. The manner in which the axes of the arguments ⍺ and ⍵ contribute to the axes of the result are summarized in Figure 6–3.

[4] To preserve the conventional order of coefficients, with the high-order term first and the constant last, the order may be reversed, making the expression `⍺⌹p∘.*⌽0,⍳d`.

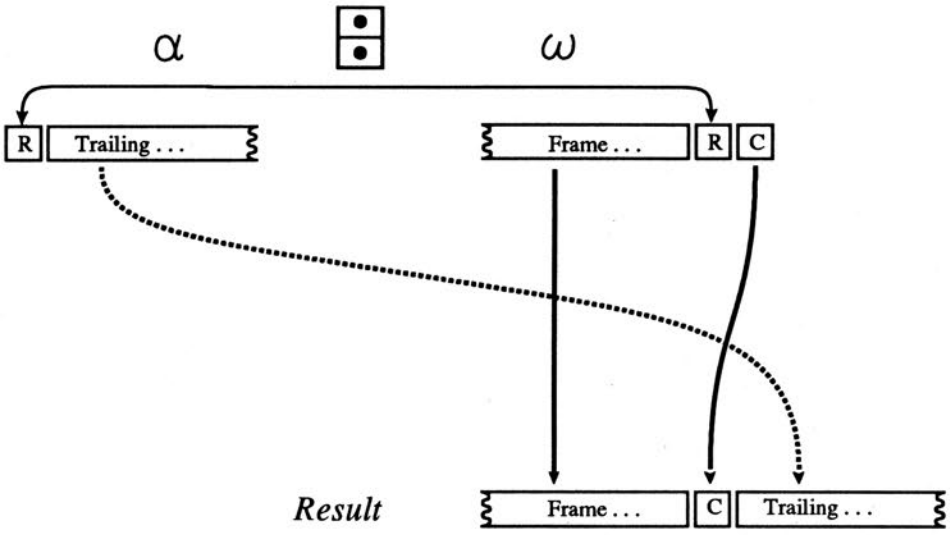


Figure 6–3 : Shape of the argument and result of dyad ⍰.

⋆ ⍟ Power, Log

	Monad	***Dyad***	*Identity Element*	*Argument Rank*
⋆	Exponential	Power; Involution	1	0 0 0
⍟	Natural log	Log base-⍺	None	0 0 0

Monads ⋆ and ⍟

The *exponential* denoted by ⋆⍵ is equivalent to e^{ω}, where e is the base of the natural logarithms, given approximately by:

```
      ⋆1
2.718281828
```

The *natural logarithm* ⍟ is inverse to ⋆ in the sense that

⍵ ⟺ ⍟⋆⍵ ⟺ ⋆⍟⍵

Moreover, if ⍷ is e, the base of the natural logarithms ⍷←⋆1,

⍟⍵ ⟺ ⍷⍟⍵.

For both real and complex arguments (except when ω is 0), the log of ω is equal to the log of the absolute value of ω plus *i* times the phase of ω; that is:

```
⊛ω  ⟷  (⊛|ω) + 0j1×12○ω
```

However, for a positive real argument, `0j1×12○ω` is zero, and so has no effect on the result. For a negative real argument, the imaginary part of the result is π radians. That is because the log is equal to the arc of ω, and the arc of any negative number is π radians. `⊛0` is undefined and results in a *domain error*.

```
      ⊛2 3,(*2), 1
0.6931471806 1.098612289 2 0

      ⊛ 2 2⍴ - 1,(*1),2,(*2)
           0j3.141592654 1j3.141592654
0.6931471806j3.141592654 2j3.141592654

      ⊛0.5 0.6
¯0.6931471806 ¯0.5108256238
```

Dyad ⋆

The expression α⋆ω is often read as "α *to the power* ω." `a⋆2` and `a⋆3` and `a⋆0.5` are respectively the square, cube, and square root of *a*. The general definition of α⋆ω, for real and complex arguments, is:

α⋆ω	⟷	⋆ω×⊛α	When α≠0
α⋆ω	⟷	0	When α=0 and ω≠0
α⋆ω	⟷	1	When α=0 and ω=0

For the simple case of a non-negative integer right argument, α⋆ω is equivalent to ×/ωρα. Since ×/ applied to an empty list yields 1, α⋆0 is 1 for any α, even when α is zero.

Raising a number to a fractional power is equivalent to extracting a root; for instance, the three-fifths root of α can be written α⋆3÷5 or the square root of α as α⋆÷2.

When α is negative and ω is fractional, the result is complex. Multiple roots may exist, some real and some complex. The result is the *first* root (proceeding counterclockwise from the positive real axis), regardless of whether it lies on the real axis. For odd integral roots, this result may sometimes be surprising. For example, although ¯2 is a cube root of ¯8, it is not the principal root by the rule just stated:

```
      ¯8*÷3
1j1.732050807568877
      1j1.732050807568877 ¯2 *3
¯8 ¯8
```

Dyad ⍟

The base-b logarithm $b⍟\omega$ is inverse to power in the sense that:

$$\omega \quad \Longleftrightarrow \quad b⍟b*\omega \quad \Longleftrightarrow \quad b*b⍟\omega$$

The dyadic logarithm $\alpha⍟\omega$ is equivalent to the ratio of the logs of the arguments considered separately; that is:

$$\alpha⍟\omega \quad \Longleftrightarrow \quad (⍟\omega)÷⍟\alpha$$

Since the logarithm of `1` to any base is `0`, `1⍟`ω is normally a *domain error* and α`⍟1` is normally `0`. But `1⍟1` is `1` since in APL `0÷0` is defined to be `1`. If either of α or ω is negative, the logarithm may be complex:

```
      ω∘.⍟ω←¯2 2 10,○1
             1 0.04642j¯0.21039 0.076663j¯0.34746
      1j4.5324                1            1.6515
0.60551j2.7444          0.60551                 1
```

○

Circle Verbs

	Monad	***Dyad***	*Identity Element*	*Argument Rank*
○	π Times	Trigonometric Functions	[none]	0 0 0

Monad ○

The expression ○ω is equivalent to π times ω, where π is the ratio of the circumference of a circle to its diameter, approximately:

```
      ○1
3.1415926535
```

Dyad ○

In the expression α○ω, the left argument α must be an integer from the set ¯12 through 12, inclusive. The value of α selects one of the families of circular verbs, as follows.

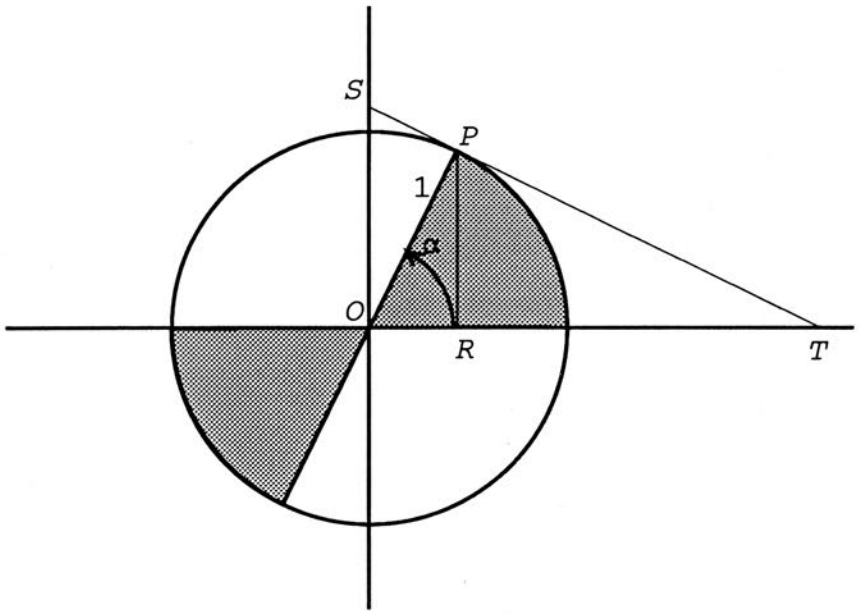
α ∈ 1 2 3	Circular or trigonometric (argument in radians)
α ∈ ¯1 ¯2 ¯3	Inverse circular or trigonometric (result in radians)
α ∈ 5 6 7	Hyperbolic
α ∈ ¯5 ¯6 ¯7	Inverse hyperbolic
α ∈ 0 4 ¯4 8 ¯8	Pythagorean
α ∈ 9 11	Complex (Cartesian coordinates)
α ∈ ¯9 ¯11	Inverse Cartesian
α ∈ 10 12	Complex (polar; arc in radians)
α ∈ ¯10 ¯12	Inverse polar

Selection of Function by Left Argument of ○

0○ω	(1−ω*2)*0.5			
1○ω 2○ω 3○ω	sin ω cos ω tan ω		¯1○ω ¯2○ω ¯3○ω	arcsin ω arccos ω arctan ω
4○ω	(1+ω*2)*0.5		¯4○ω	ω×(1−ω*¯2)*0.5
5○ω 6○ω 7○ω	sinh ω cosh ω tanh ω		¯5○ω ¯6○ω ¯7○ω	arsinh ω arcosh ω artanh ω
8○ω	(¯1−ω*2)*0.5		¯8○ω	−(¯1−ω*2)*0.5
9○ω 10○ω 11○ω 12○ω	Real part: (ω++ω)÷2 Magnitude: |ω Imaginary: (ω−+ω)÷0j2 Phase angle: 11○⊗ω		¯9○ω ¯10○ω ¯11○ω ¯12○ω	Identity +ω 0j1×ω *0j1×ω

Geometric View of the Pythagorean and Circular Functions

The various circle verbs can be represented by constructions on a unit circle whose origin is *O* (see Figure 6−4). The distance *OP* is 1. The angle *a* establishes the point *P* at which a ray through the origin cuts the unit circle. For the angle *a*, the altitude *PR* represents the sine of *a*, *OR* the cosine of *a*, and *PT* the tangent of *a*.



	PR sine *a*
OP = 1	*OR* cosine *a*
PR = 0○*OR*	*PT* tangent *a*
OR = 0○*PR*	*PS* cotangent *a*
OT = 4○*PT*	*OT* secant *a*
PT = ¯4○*OT*	*OS* cosecant *a*

Figure 6–4 : Pythagorean and circular verbs in terms of the unit circle.

The inverse functions ¯1○ and ¯2○ have domains restricted to values whose magnitude does not exceed 1. Since sine and cosine are cyclic, a given sine or cosine is related to any of an infinite number of arcs, differing from each other by successive multiples of 2π. For a given sine, ¯1○ returns the arc lying at or between the range $-\frac{\pi}{2}$ and $\frac{\pi}{2}$ radians.

Geometric View of the Hyperbolic Functions

Figure 6–5 extends the geometric constructions shown in the preceding figure. Here again *O* marks the origin, the center of a unit circle, so that the distance *OP* is 1. The two arms of the unit hyperbola are shown tangent to the unit circle, with the major axis of the hyperbola along the horizontal axis of the figure. As before, the line *SPT* is tangent to the unit circle.

The hyperbolic cosine relates the coordinates of a point on the unit hyperbola to the sum of the two areas (shaded in Figure 6–5) each bounded by the following three curves:

- A line from the origin to that point.

- The unit hyperbola itself.
- The major axis of the unit hyperbola. In the figure, `Q` is such a point. If (as shown here) the unit hyperbola is oriented so that its major axis lies horizontally, the hyperbolic sine is the vertical distance from the origin to `Q` (equal to `TQ`). The hyperbolic cosine is the horizontal distance `OT`. The hyperbolic tangent (by analogy with the circular tangent) is the ratio of the hyperbolic sine to the hyperbolic cosine.

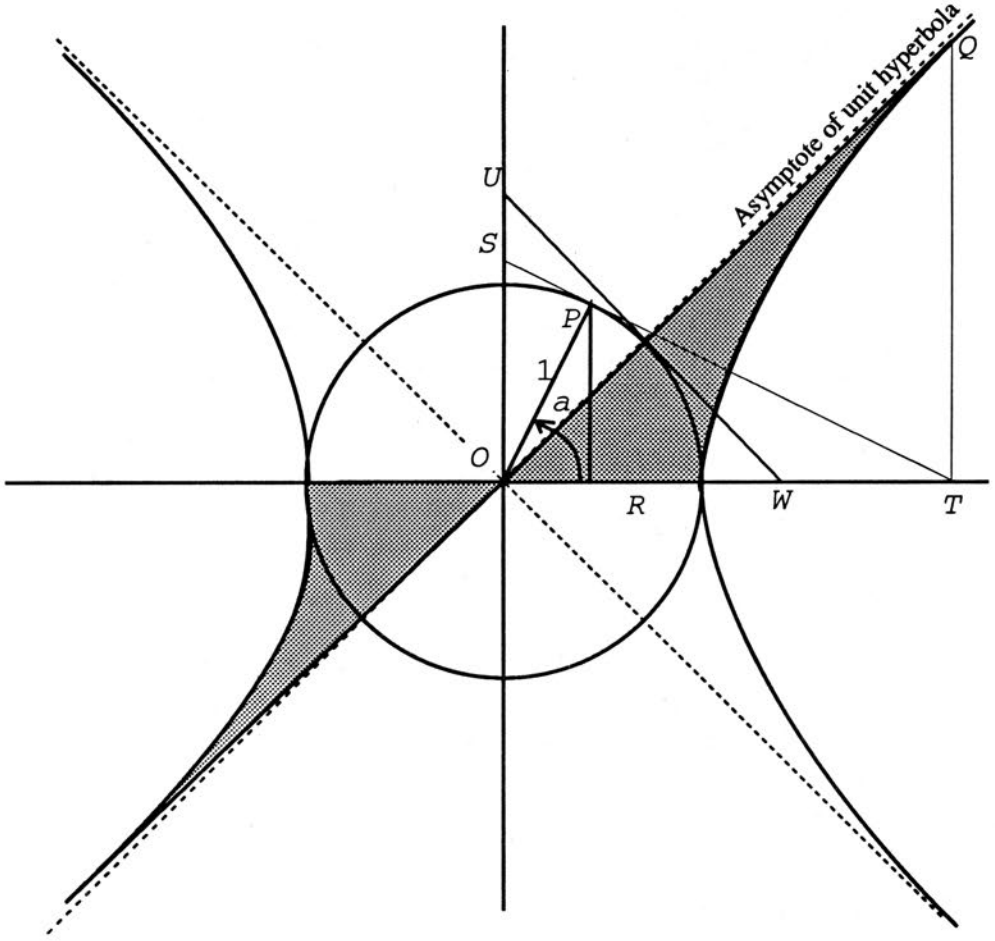


Figure 6–5 : Hyperbolic verbs in terms of the unit hyperbola.

`RP`	`=`	`1○a`	sine	`TQ` = `5○b`	hyperbolic sine
`OR`	`=`	`2○a`	cosine	`OT` = `6○b`	hyperbolic cosine
`PT`	`=`	`3○a`	tangent	`VW` = `7○b`	hyperbolic tangent
`OS`	`=`	`÷1○a`	cosecant	`PS` = `÷5○b`	hyperbolic cosecant
`OT`	`=`	`÷2○a`	secant	`OR` = `÷6○b`	hyperbolic secant
`PS`	`=`	`÷3○a`	cotangent	`UV` = `÷7○b`	hyperbolic cotangent
				`b` = `¯5○3○a`	"Gudermannian"

The inverse hyperbolic functions are *arsinh, arcosh,* and *artanh.* The "ar" prefix indicates "area" (whereas the prefix "arc" used with the inverse circular functions indicates "arc").

The inverse hyperbolic functions give you twice the area of the hyperbolic sector described by the corresponding *sinh, cosh,* or *tanh.* If x is a value of the hyperbolic sine and y is the value of the corresponding hyperbolic cosine, and z is their ratio, then the area enclosed is equal to:

`¯5○x` which equals `¯6○y` which equals `¯7○z`

The result returned for the area of the hyperbolic sine and cosine has the same sign as its argument. Since in principle *sinh* and *cosh* can take on any values, both the domain and the range of *arsinh* and *arcosh* are infinite. Since *sinh* must always be less than the corresponding value of *cosh, tanh* has a magnitude that approaches but can never reach 1.

In practice, the areas of the hyperbolic functions are evaluated by exploiting the following identities:

```
¯5○ω  ⟺  ⊛ω+ 4○ω
¯6○ω  ⟺  ⊛ω+¯4○ω
¯7○ω  ⟺  0.5×⊛(1+ω)÷(1−ω)
```

This method of computation gives `¯5○` and `¯6○` an effective domain of all numbers whose magnitude is less than about 10^{37}; the results have magnitudes less than about 85. The *artanh* function `¯7○` has an effective domain of numbers whose magnitude is less than about 18.

Representations of Complex Numbers

A complex number may be resolved to its real and imaginary parts by the expression `9 11 ∘.○ ω`; for example:

```
      9 11∘.○3j4
3 4
```

Or, it may be resolved into its magnitude and phase by `10 12∘.○ω`; for example:

```
      10 12∘.○3j4
5 0.927295218
```

You get the inverse transformations by exploiting the identities:

```
ω ⟷  ¯9 ¯11 +.○  9 11 ∘.○ ω
ω ⟷ ¯10 ¯12 ×.○ 10 12 ∘.○ ω
```

(See also the discussion of complex arguments accompanying the descriptions of + − × ÷.)

⌈ ⌊
Floor, Ceiling; Max, Min

	Monad	***Dyad***	*Identity Element*	*Argument Rank*
⌈	Ceiling	Maximum	$-\infty$	0 0 0
⌊	Floor	Minimum	∞	0 0 0

Monads ⌊ and ⌈

The expression ⌊ω returns the *floor* or *integer part* of ω. When ω is an integer, the result is ω; but when ω has a fractional part, the result is the first integer that is smaller than ω. Stating it the other way, ⌊ω is the largest integer that is not greater than ω.

```
      ⌊3.1 0 ¯6.7 ¯9
3 0 ¯7 ¯9
```

The expression ⌈ω returns the *ceiling* of ω. When ω is an integer, the result is ω; but when ω has a fractional part, the result is the next larger integer. Thus, ⌈ω is the smallest integer that is not smaller than ω.

```
      ⌈3.1 0 ¯6.7 ¯9
4 0 ¯6 ¯9
```

⌈ω can be defined in terms of ⌊ω thus:

```
⌈ω ⟷ ⌊⍥-⍥-ω ⟷ -⌊-ω
```

Tolerant Floor and Ceiling

The implied comparison of ω with the integers is *tolerant.* When ω is sufficiently close to an integer, both $\lceil\omega$ and $\lfloor\omega$ are ω. "Sufficiently close" is defined in terms of the visible value of the system variable `⎕ct`. The formal definitions of floor and ceiling are:[5]

```
floor: n-(⎕ct×1⌈|ω)<((n←(×ω)×⌊0.5+|ω)-ω

ceiling: -floor-ω
```

Thus, if x is any integer+`⎕ct`, the ceiling of x is that integer, even though in that case $\lceil x$ is (strictly speaking) smaller than x.

The default value of `⎕ct` is `1.4196976927394189e¯14`.

Floor of a Complex Number

You can represent a complex number as a geometrical point in the complex plane: a plane in which displacement along the x-axis represents the real portion and displacement along the y-axis represents the imaginary portion of a number. The floor of a complex number lies at a point on the intersection of lines for real and imaginary integers; that is, both its real and its imaginary parts are integers. You can think of the process of finding the floor as motion from a point anywhere in the complex plane to a point at one of the vertices of the surrounding square bounded by the real and imaginary integer lines (see Figure 6–6).

The vertex chosen is one that satisfies the constraint that the distance between a number and its floor is always less than 1. Thus, there is a zone in each square for which the floor is *not* what you would get by taking the floors of the real and imaginary parts separately.[6] The algorithm by which SHARP APL finds the floor of a complex number is the one proposed by McDonnell in 1973.[7] Imagine a grid of integer lines on the complex plane. The task is to map a point anywhere in the plane to one of the lattice points on the grid – that is, to a point whose coordinates are described by a real integer and an imaginary integer. The floor `f` of a complex number ω is found by the following algorithm.

- Let `LL` be the point at the lower left corner of the unit square containing ω.

---

5 These were derived by Robert Bernecky and D.L. Forkes in 1978. See *SHARP APL Technical Note 23: Comparison Tolerance*, I.P. Sharp Associates, 1978.

6 An application that requires complex floor calculated in that way can obtain it by `¯9 ¯11+.○⌊9 11∘.○ω`.

7 Eugene E. McDonnell, *Integer functions of complex numbers with applications*, IBM Scientific Centers Technical Report No. 320-3015, 1973. Summarized in: Eugene E. McDonnell, "Complex Floor," in Per Gjerløv, ed., *APL Congress 1973*, North Holland Publishing, 1973.

- Let *fr* and *fi* be the fractional parts of the real and imaginary parts of ⍵.

  Then,

  When 1>*fr*+*fi* : *f*←*LL*

  When (1≤*fr*+*fi*) *and* *fr*≥*fi* : *f*←*LL*+1

  When (1≤*fr*+*fi*) *and* *fr*<*fi* : *f*←*LL*+0*j*1

Those rules can be stated as APL direct definitions in which ⌊ and | are applied only to real arguments:

```
floor: (LL ⍵) + ((case1 ⍵),(case2 ⍵),case3 ⍵)/0 0j1 1
   LL: (⌊real ⍵) + 0j1×⌊imag ⍵
case1:  1>(fr ⍵)+fi ⍵
case2: (1≤(fr ⍵)+fi ⍵) ∧ (fr ⍵)< fi ⍵
case3: (1≤(fr ⍵)+fi ⍵) ∧ (fr ⍵)≥ fi ⍵
   fr: 1|real ⍵
   fi: 1|imag ⍵
 real: 9○⍵
 imag: 11○⍵
```

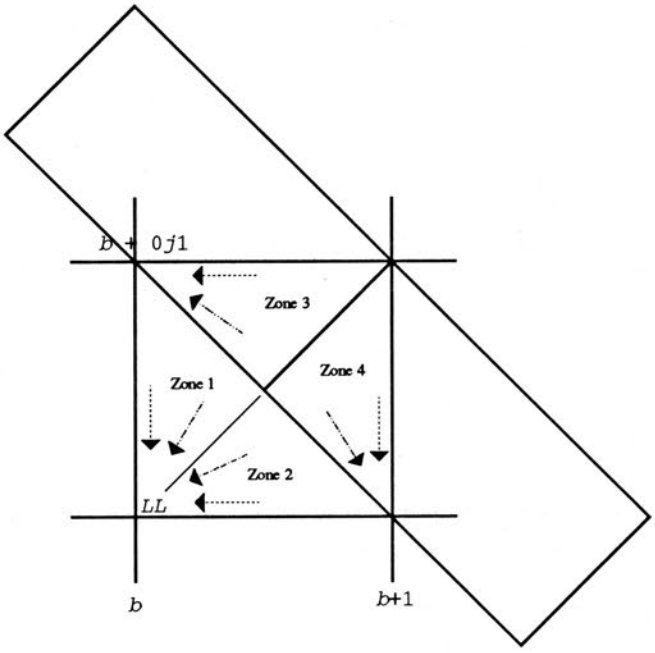


Figure 6–6 : Mapping of points in a unit square to complex integers.

In Figure 6–6, a unit square is divided by its diagonals into four zones. A point in Zone 1 or Zone 2 is mapped to *LL*; a point in Zone 3 to *LL*+0j1, and a point in Zone 4 to *LL*+1.

The zone of all points in the plane that are mapped to a particular complex integer is thus a rectangle tilted at 45°, with the floor at the center of its lower edge. Figure 6–7 shows a portion of the complex plane tiled by such rectangles.

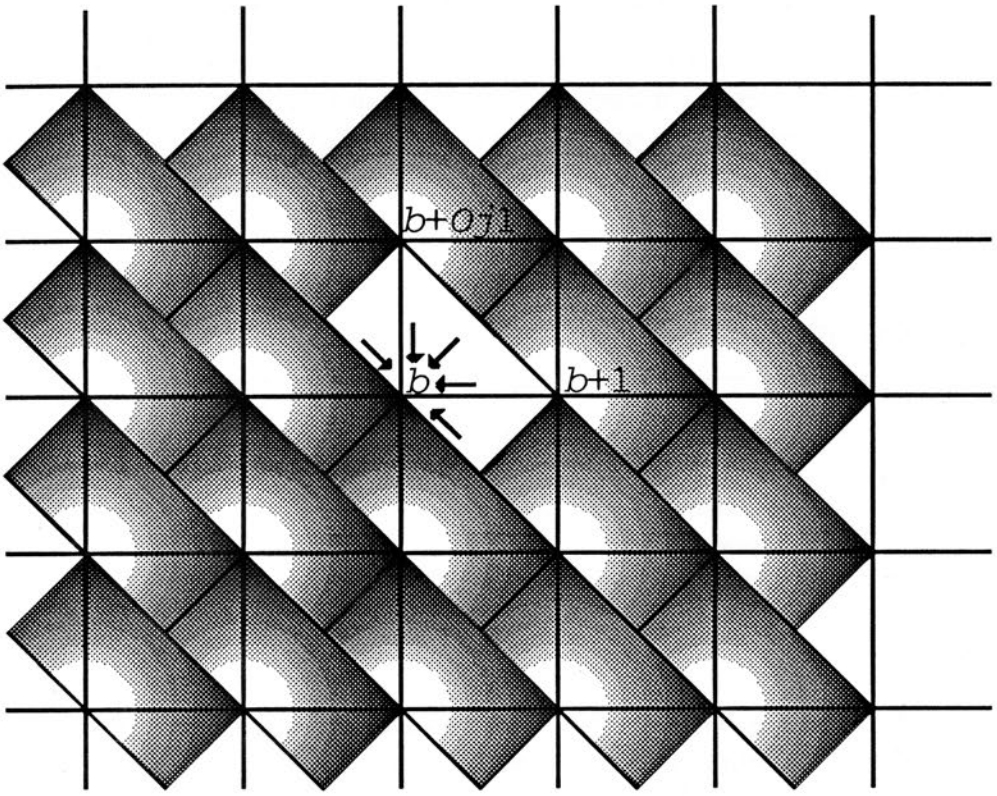


Figure 6–7: The complex plane tiled by areas having equal floors.

Dyads ⌊ and ⌈

$\alpha\lfloor\omega$ returns the minimum of α and ω (that is, whichever is smaller), and $\alpha\lceil\omega$ returns the maximum (that is, whichever is greater). For example:

```
      3⌊4 ¯4
3 ¯4
```

Maximum and minimum are not tolerant; that is, their results are not subject to □*ct*. These verbs are defined only for real arguments; the system reports a *domain error* when an argument is complex or non-numeric.

When the reduction adverb ⌿ modifies ⌈, the result is the maximum of the major cells of ⍵. For example, if the shape of ⍵ is `3 4 5`, then

```
⌈⌿⍵  ⟺  ⍵[1;;] ⌈ ⍵[2;;] ⌈ ⍵[3;;]
```

When the first axis of ⍵ has length zero, the result is the identity element for maximum. That is the number compared to which all other numbers are bigger: minus infinity. In practice, the APL system returns the smallest representable value. The actual value depends on the particular system. In SHARP APL, it is approximately equal to -7.237×10^{75}.

For the same reason, the identity element for ⌊ is positive infinity, and the system returns the largest number it can represent. Here too, the actual value depends on the particular system. In SHARP APL, it is approximately 7.237×10^{75}.

Magnitude; Residue

	Monad	***Dyad***	*Identity Element*	*Argument Rank*
|	Magnitude; Absolute value	Residue	0	0 0 0

Monad |

The definition `|⍵ ⟺ (⍵×+⍵)*0.5` yields the *magnitude* or *absolute value* of ⍵. This definition applies equally to real and to complex arguments. For a complex ⍵, the result is the hypotenuse of a right triangle whose sides are the real and imaginary parts of the number. If the real and imaginary parts of ⍵ are denoted as `re` and `im`, then `|⍵` is `((re*2)+im*2)*0.5`.

Dyad |

A familiar use of residue is to determine the remainder resulting from dividing a non-negative integer by a positive integer. For example:

```
      3|0 1 2 3 4 5 6 7 8.1
0 1 2 0 1 2 0 1 2.1
```

The definition `⍺|⍵ ⟺ ⍵-⍺×⌊⍵÷⍺+0=⍺` extends this notion to a zero left argument. When ⍺ is zero, the result is the right argument unchanged. It also extends the verb's domain to fractional right arguments and to negative

or fractional left arguments. When α is a negative integer, the result ranges between α and zero, just as it does when α is positive. For example:

```
      ¯3|¯4 ¯3 ¯2 ¯1 0 1 2 3 4
¯1 0 ¯2 ¯1 0 ¯2 ¯1 0 ¯2

      1|2.5 3.64 2 ¯1.6
0.5 0.64 0 0.4
```

However, for cases such as `(÷3)|2÷3`, in order to produce a true zero (rather than a small fraction) the residue is made *tolerant* in the following way. Where α is non-zero, and ω is not an integer multiple of α, the residue is found by subtracting from ω the product $\alpha\times S$, where S is the tolerant floor of $\omega\div\alpha$. Stating the same thing more formally:

$$\begin{aligned} &\text{Let} && S\leftarrow\omega\div\alpha+\alpha=0 \\ &\text{When } (\alpha\neq 0)\wedge(\lceil S)\neq\lfloor S, && \alpha\mid\omega \iff \omega-\alpha\times\lfloor S \\ &\text{Otherwise,} && \alpha\mid\omega \iff \omega\times\alpha=0 \end{aligned}$$

For example:

```
      0.1|2.5 3.64 2 ¯1.6
0 0.04 0 0
```

Consider the result when the modulus is `3` or `¯3` and the right argument is `10` or `¯8`:

```
      3  ¯3 ∘.| 10 ¯8
 1  1
¯2 ¯2
```

Figure 6–8 illustrates the results obtained from the above example.

The definition of residue applies equally to complex arguments. The extension takes advantage of the properties of `⌊` (*floor*) applied to complex arguments. As a consequence, the residue of any number (whether real or complex) is always less in magnitude than the modulus (except for the special case of zero modulus, when the residue has the same value as the right argument).

For example, when `⎕ct` is set to `0`, because of imprecisions in the internal representation, the following expression misses certain multiples:

```
      0.24j¯.32|⍳5 ⊣⎕ct←0
0.12j¯0.16 0.32j0.24 0.12j¯0.16 0.32j0.24 0.12j¯0.16
```

But with the default value of `⎕ct`, SHARP APL is able (correctly) to detect that `2` and `4` are multiples of `.24j¯.32` and to return a `0` residue:

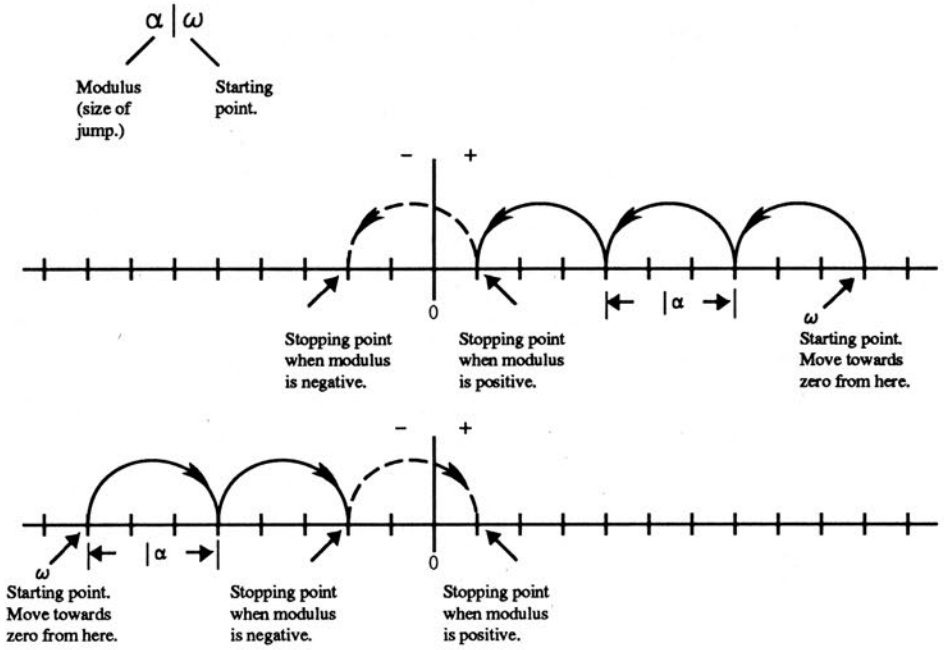


Figure 6–8: Schematic representation of `3 ¯3|10` *and* `3 ¯3|¯8`.

```
      )clear
clear ws
      0.24j¯0.32 |⍳5
0.12j¯0.16 0 0.12j¯0.16 0 0.12j¯0.16
```

!

Factorial; Combinations

	Monad	***Dyad***	*Identity Element*	*Argument Rank*
!	Factorial; Gamma	Out of; Combinations; Binomial coefficients	1	0 0 0

Monad !

When ⍵ is a non-negative integer, the factorial is defined by:

```
!⍵ ⟺ ×/⍳⍵
```

For example, `!4` is `×/1 2 3 4`, or `24`, and `!0` is `1`.

For a negative integer, the factorial is undefined. For a real argument other than a non-negative integer, `!⍵` is closely related to the *gamma* Γ function,[8] so that a value of the gamma function can be obtained from:

$$\Gamma(\omega) \quad \Longleftrightarrow \quad !\omega-1$$

Dyad !

The dyad `⍺!⍵` is defined in terms of the monad thus:

```
⍺!⍵  ⟺  (!⍵)÷(!⍺)×(!⍵-⍺)
```

For non-negative integer arguments, `⍺!⍵` yields the number of distinct ways of selecting `⍺` things from `⍵` things; this accounts for the name "out of," and for its use in producing binomial coefficients. For example:

```
      0 1 2 3 4!4
1 4 6 4 1
```

For a negative integer `I`, the expression `!I` is not defined, because, near a negative integer, the magnitude of `!I` approaches infinity. In the definition of `⍺!⍵` shown above, these near-infinities may cancel, permitting `⍺!⍵` to be evaluated provided we assume that these infinite values occur in the following ways:

1. When `⍺` and `⍵` are both non-negative, but `⍺>⍵`, the term `!⍵-⍺` is infinite, yielding `0` for the result of `⍺!⍵`. This agrees with the observation that there are no ways to pick `⍺` things from a smaller collection.

2. When infinite values occur in both numerator and denominator, they are assumed to cancel. This can be seen in the values in the following function table:

```
      i←¯3 ¯2 ¯1 0 1 2 3
      i ∘.! i
 1  ¯2   1   0   0   0   0
 0   1  ¯1   0   0   0   0
 0   0   1   0   0   0   0
 1   1   1   1   1   1   1
¯3  ¯2  ¯1   0   1   2   3
 6   3   1   0   0   1   3
¯10 ¯4  ¯1   0   0   0   1
```

[8] The gamma function (and also the beta function B, mentioned on the next page) are outside the scope of this manual. See a standard mathematical reference, for example *Handbook of Mathematical Functions*, US Department of Commerce, National Bureau of Standards, Applied Mathematics Series #55.

The generalized definition adopted for dyad ! permits evaluation of the Beta function by the identity:

```
B(α,ω)  ⟺  ÷ω×(α-1)!α+ω-1
```

?

Roll; Deal

	Monad	*Dyad*	*Identity Element*	*Argument Rank*
?	Roll	Deal	[none]	0 ⋆ ⋆

The *random* verb ? selects either a single number or a set of distinct numbers from a population of numbers. For example, each of the following selects from the first ten integers:

```
      ?10
4

       3?10
9 2 7
```

The size of the population from which numbers are selected is determined by the value of the right argument. The population is the first ω integers; that is, a result cell is selected from ⍳ω. The population is therefore affected by the value of ⎕io.

Monad ?

The verb *roll* is named from the analogy with rolling a die to choose one of a set of numbers with equal probability.

Dyad ?

The verb *deal* is named by analogy with dealing from a pack of cards. The result of α?ω for each of the corresponding cells of α and ω is a list of length α in which all the elements are distinct.

Argument Rank

The default rank of dyad ? is at present undefined; ? may be used with array arguments only by stating its argument rank explicitly. For example, to independently generate four random numbers taken from ⍳100 and also four taken from ⍳200:

```
      4 ?⍤⍤0 ⊢ 100 200
56  27  84  20
20 116 133 180
```

Pseudo-random Algorithm and □*rl*

The results of *?* (both monad and dyad) are "pseudo-random." That is, although the results (collectively) satisfy most tests of randomness, they are produced by an explicit and repeatable algorithm. The value returned depends on the visible value of the system noun □*rl* (*random link*). Each time SHARP APL (successfully) executes *?*, it sets □*rl* to a new value. If, on different occasions, you set the same value of □*rl* and then invoke *?* on the same arguments in the same sequence, you get the same results. (See the discussion of □*rl* in Chapter 16, "System Nouns and Verbs".)

The algorithm for generating random numbers is the *Linear Congruential Method* developed by D.H. Lehmer in 1951.[9] The algorithm depends on two constants: a prime *p*←2147483647 and a multiplier *m*←16807. (The prime is the largest prime representable with 31 binary digits. The multiplier is a primitive root of *p* that is close to the square root of *p*; the value 16807 is 7*5.)

When *ω* is an item less than 2^{31}, SHARP APL finds the value of *?ω* by

```
?ω ⟺ □io←⌊ω×□rl÷p ⊣ □rl←p|m×□rl
```

in which ⌊ means the non-tolerant floor (as one would get when □*ct* is 0).

For an argument greater than or equal to 2^{31}, SHARP APL sets □*rl* twice. If the result of the intermediate set is called *rl*, so that

```
□rl←p|m×rl←p|m×□rl
```

the formula becomes

```
?ω ⟺ ⌊□io←ω×((rl←□rl×2*32)÷16*8)÷¯1+2*31
```

Random numbers generated in this fashion are rectangularly (or "uniformly") distributed; you can get other distributions by applying various transformations to the rectangular values that *?* generates. For example, random normal deviates with *mean*=0 and σ=1 may be obtained by:

```
normal: (1○2×uniform ω)×(¯2×⍟uniform ω)*0.5
uniform: (?ω⍴m←2147483647)÷m ⊣□io←1
```

SHARP APL evaluates *α?ω* using repeated calls to a short cut version of the mechanism for monad *?*. In principle, the method is to generate

[9] See D.E. Knuth, *The Art of Computer Programming – Seminumerical Algorithms*, Vol. 2, Addison-Wesley, 1969.

random integers, keeping those that fall within the range specified by ⍺ and discarding those that do not. However, the result of dyad ? is not necessarily what you would get by using monad ? and discarding out-of-range items from the result.

< > ⊃
Box, Open; Link

	Monad	***Dyad***	*Identity Element*	*Argument Rank*
<	Box; Enclose	[see Less]	[none]	∞ 0 0
>	Open; Disclose	[see Greater]	[none]	0 0 0
⊃	Conditional box	Link	[none]	∞ ∞ ∞

The monads < and > form a box and open a box, respectively. (The dyads < and > are the relations *less* and *greater*, discussed in the next section.)

A box is an item (or scalar). It has no axes. However, it may contain an array of any type, rank, or shape. A box is thus a scalar encoding of an array.

Because a box is an item, within an array of boxes each individual box occupies a single position, just as a number occupies a single position in a numeric array, or a character in an array of characters. An array of boxes thus permits you to handle nouns of arbitrary rank and shape within the same simple rectangular framework used for numbers or characters.

Boxing an array always produces a result that is an item and always increases by 1 the number of times it will be necessary to open the item to retrieve its unboxed contents. This is made obvious when you set ⎕*ps* so that the boundaries of a box are shown explicitly in its display. For example:

```
      ⎕ps←0 0 ¯2 ¯2
      ⊢x←126
126                                  ⍝Open item

      <x                             ⍝One level of boxing
|¯¯¯|
|126|
|___|
```

```
      <<x                          ⍝Two levels of boxing
┌─────┐
│┌───┐│
│││126│││
│└───┘│
└─────┘
```

Thus, when the argument ω is already an item, $<\omega$ boxes it again and produces a result with an additional level of boxing; $<\omega$ is never identical to ω.[10] Thus, an open item (that is, a number or character) is not equal to any boxed item, not even to the same item enclosed in a box:

```
      2 ≡ <2
0
```

Monad >

The monad > is the inverse of <: it opens a box, to disclose what lies within. When ω is an item, $>\omega$ returns the array that was boxed to produce ω.

When the argument ω is already open, the result is the same as the argument.

An attempt to open arrays containing mixed character and numeric data results in a *domain error*.

Since > has argument-rank 0, all the axes of ω are *frame* axes. Thus, the result returned by $>\omega$ has the same axes as ω, followed by the axes of the arrays disclosed from the various boxes. When all the boxes contain cells of the same rank and shape, the cell axes returned by $>\omega$ are the axes common to all of them. For example, when ω is a 5-by-2 table of boxes and each box contains a 3-by-4-by-7 array, then the shape of $>\omega$ is `5 2 3 4 7`.

When each item is boxed, items of irregular type or shape can be arranged in a regular array. Thus, *open* may be usefully modified by the *under* conjunction ⍤⍤, using open with another verb which is to be applied to each item of the boxed argument(s), such as *lengths* `↑⍤⍤>` *pieces*.

Tolerant Open

When the arrays contained in the various boxes are of the same type, but differ in rank or shape, each result is "padded" as necessary so that every

[10] In this regard, the rules for boxing in SHARP APL differ from those governing the analogous verb in APL2. APL2 does not permit an item to be boxed, but returns the same item unchanged. Since APL2 restricts boxing to arrays whose rank is greater than 0, in order to have items and boxes in the same array, APL2 *requires mixed arrays*, in which the higher-rank constituents are boxed but the items are not.

cell in the result has the same rank and shape. In this sense, *open* is said to be *tolerant*, since it does not require the various arrays to have the same shape. Each is given a rank equal to the maximum rank boxed within ⍵ by logically catenating leading 1s to its shape vector. Each axis is given a length equal to the maximum length that that axis has in any of the boxes in ⍵:

```
      ⊢⍵←¯1⍤<' The quick brown fox.'
┌───┐┌─────┐┌─────┐┌────┐
│The│|quick|│brown│|fox.|
└───┘└─────┘└─────┘└────┘

      >⍵
The
quick
brown
fox.

      ⍴>⍵
4 5
```

Fill Elements

Where the array within a box has a length less than the maximum length for that axis, the corresponding cell in the result is padded with *fill elements* – extra items to make up the needed length. A character array is padded with trailing blanks, a numeric array with 0s, and an array of boxes with the enclosed empty box <⍳0.

Each cell of the result of >⍵ is expanded to have the common shape of all the result cells, as follows:

Cell – the array disclosed by opening a single box in ⍵
MaxRank – highest rank inside all boxes in ⍵
MaxShape – maximum length of the axes inside all boxes in ⍵

CommonShape: `MaxShape↑(1⌈(-MaxRank)↑⍴Cell)⍴Cell`

This is illustrated in Figure 6–9.

When you use one of the composition adverbs (described in Chapter 7, "Adverbs and Conjunctions"), it is possible to combine *open* with *box* for each cell. Each box in the argument is opened, processed in some manner, and then re-boxed. In that situation, there is then no need for padding,

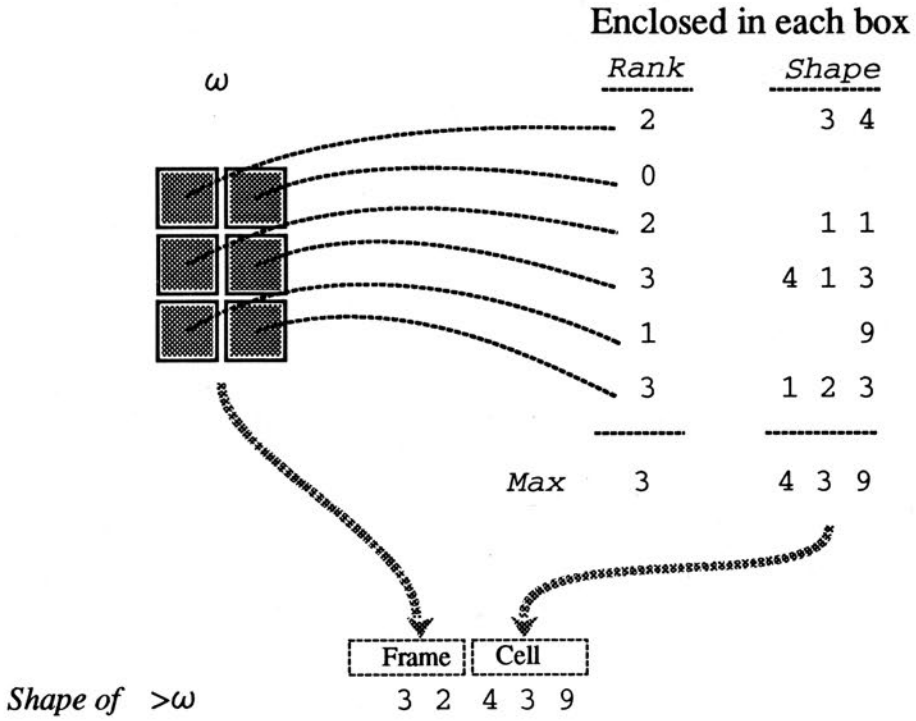


Figure 6–9: Shape of result cells produced by tolerant open.

because the variously-shaped intermediate results are not assembled into the result until re-boxed.[11]

Monad ⊃

When ω consists of boxes, the result of monad ⊃ is ω, unchanged. But when ω is open, ⊃ω boxes it. This is why monad ⊃ is called *conditional box*.

You can use the expression ω≡⊃ω to determine whether ω is open or boxed.

Dyad ⊃

The dyad ⊃ (*link*) forms its arguments into an array of boxes. It always boxes α, but it boxes ω only when ω is open.

Thus, dyad ⊃ can be defined in terms of dyad join , and monad ⊃ as follows:

```
α⊃ω  ⟺  (<α),⊃ω
```

11 In SHARP APL, the special case expression α f⍤0 ω corresponds to the APL2 *each* expression α f¨ ω.

The *conditional boxing* of the right argument lets you use a simple expression to box and join several objects without parentheses:[12]

```
      Who←'Tom'⊃'Dick'⊃'Harry'
      ⍴Who
3

      Who
┌───┐┌────┐┌─────┐
│Tom│|Dick||Harry|
└───┘└────┘└─────┘

      ⊢Who←'Jane'⊃Who
┌────┐┌───┐┌────┐┌─────┐
|Jane||Tom||Dick||Harry|
└────┘└───┘└────┘└─────┘
```

Using link with the rank conjunction lets you link one array with parts of another. Consider a compiler that generates declarations for another computer language for a list of names. If *names* is a list of boxed names, then the declarations might be generated as follows:

```
      ⍪names
i
foo
vtom
squeeze
nub
      'static VAR '⊃⍤1 0 names
static VAR i
static VAR foo
static VAR vtom
static VAR squeeze
static VAR nub
```

Watch out: When you need to add another box at the end of *Who* – for example, to add *'Jill'* so that her name follows the other names – it is *not* appropriate to write:

```
      Who←Who⊃'Jill'
```

That would put a new level of boxing around *Who*, and result in the two-item list:

[12] This achieves the principal benefit of APL2's *vector* or *strand notation* without the syntactic complexities that APL2 introduces by allowing nouns to be linked simply by juxtaposition.

```
|----------------------|
||----||---||----||-----|||----|
||Jane||Tom||Dick||Harry|||Jill|
||____||___||____||_____|||____|
|______________________|
```

To add one more box at the end:

```
Who←Who,<'Jill'
```

< >

Less; Greater

≤

Less than or Equal

≥

Greater than or Equal

	Dyad	*Identity Element*	*Argument Rank*
<	Less	0	∞ 0 0
≤	Less than or equal	1	* 0 0
≥	Greater than or equal	1	* 0 0
>	Greater	0	0 0 0

These four dyads are discussed together because they are closely related. There is no monad ≤ or ≥. Monads < and > are *box* and *open*, discussed in the preceding section.

A result cell returned by these propositions contains `1` where the proposition is true, and `0` otherwise. The domain is restricted to real numbers. (APL has no collating sequence for characters. Hence, comparisons such as `'a'<'b'` produce *domain error*.)

Each of these verbs makes *tolerant* comparisons, subject to the visible value of the system variable `⎕ct`. To examine the effect of `⎕ct`, let *tEqual* be a user-defined verb whose result is `1` when `⍺` and `⍵` are equal within the limits of tolerance set by `⎕ct`.[13]

```
tEqual: 0≤(⎕ct×(|⍺)⌈|⍵)-|⍺-⍵
```

[13] See the discussion of "tolerant comparison" in the section of "equal and unequal", which follows.

Then the following shows how the four propositions < ≤ ≥ > are affected by `⎕ct` by referring to expressions in which the effect of `⎕ct` is confined to `tEqual`:

```
α<ω   ⟺  (α<ω) ∧ ~α tEqual ω
α≤ω   ⟺  (α<ω) ∨   α tEqual ω
α≥ω   ⟺  (α>ω) ∨   α tEqual ω
α>ω   ⟺  (α>ω) ∧ ~α tEqual ω
```

= ≠
Nubsieve; Equal, Unequal

	Monad	***Dyad***	*Identity Element*	*Argument Rank*
=	[none]	Equal	1	⋆ 0 0
≠	Nubsieve	Unequal	0	∞ 0 0

Monad ≠

Monad ≠ is defined as *nubsieve*. It produces a Boolean list that marks the nub (the unique major cells) of ω. For example, with a list argument, the unique items are found. With a table, the unique rows of the table are found, and so on:

```
      ≠'abacus'
1 1 0 1 1 1
      b
mabel
toto
lisa
fred
lisa
fred
dorothy
toto
mabel
fred
nancy
ed
nancy
      ≠b
1 1 1 1 0 0 1 0 0 0 1 1 0
      (≠b)≠b
mabel
toto
lisa
fred
dorothy
nancy
ed
```

The commonly used expression for the nub of a list

```
nub: ((⍳⍴⍵)=⍵⍳⍵)/⍵
```

can be written more efficiently and compactly using nubsieve as

```
nub: (≠⍵)/⍵
```

Formally, nubsieve is defined on lists as:

```
ns: ⍴0 : ×⍴⍵ : (<\~a)∨a\ns(a←⍵≠1⍴⍵)/⍵
```

For arrays, major cells are used in place of list elements.

Since nubsieve is defined in terms of tolerant inequality, it is sensitive to ⎕*ct*.

Dyad =

A result cell of $\alpha=\omega$ is `1` when the items compared are equal, and `0` otherwise.

No character is equal to any number; in particular:

```
      123='123'
0 0 0
```

Two characters are equal only when their internal representations are equal (that is, when they have the same position in `⎕av`, the set of all possible characters). For example:

```
      'a'='Ananas'
0 0 1 0 1 0
```

In a few cases, usually involving characters whose appearance differs according to national convention, a device may generate the same display for two different characters. The fact that the displays are indistinguishable does not make the characters equal.

Tolerant Comparison of Real Numbers

SHARP APL judges two numbers to be equal when the absolute difference between them is not greater than `⎕ct` times the greater of their magnitudes. Formally, using the non-tolerant meaning of $\leq$:

$$(|\alpha-\omega)\leq \square ct\times(|\alpha)\lceil|\omega$$

SHARP APL limits the maximum permissible value of `⎕ct`. To be valid, `⎕ct` must be a non-negative item less than `16*¯8` (which is about $2.328306437\times10^{-10}$).[14]

There are two practical consequences of the way `⎕ct` affects comparisons and the limit on the maximum value `⎕ct` can be given:

- Tolerance cannot affect comparisons with zero. That is, no comparison in which one of the items is `0` can be judged equal to another unless they are exactly equal. This is true regardless of the value of `⎕ct`.
- Tolerant comparison can make a difference only to numbers whose internal representation requires floating point. (Stating it the other way, no matter what legal value you give `⎕ct`, two numbers whose internal type is *integer* can be reported "equal" only when they are exactly equal.)

[14] If you give `⎕ct` a value outside that range, SHARP APL does not immediately object. It reports `⎕ct error` at the next use of a verb that implicitly refers to `⎕ct`.

Dyad ≠

The verb *unequal* is the negation of *equal:*

$$\alpha \neq \omega \iff \sim \alpha = \omega$$

≡

Match

	Monad	*Dyad*	*Identity Element*	*Argument Rank*
≡	[none]	Match	[none]	⋆ ⋆ ∞ ∞

The dyad $\alpha \equiv \omega$ returns a Boolean item reporting whether α is identical to ω. To be judged identical, α and ω must have:

- the same *rank*
- the same *shape*
- Throughout that shape, the same *value.*

For numeric items, the magnitude of their difference must be no greater than `⎕ct` times the greater of their magnitudes. (See the discussion of "tolerant comparison" in the description of "equal", earlier in this chapter.)

Since each item in one array must be equal to the corresponding item in the other, arrays that are not empty must necessarily be of the same type (numeric or character). However, empty arrays of the same rank and shape match regardless of type. Moreover, the internal type of a numeric array does *not* affect comparison. For example, an array stored in the integer format may match an array stored in floating format, provided, of course, that the actual values in the floating point array are equal (or sufficiently close) to those in the integer array.

Match has infinite default argument rank. That is, unless explicitly modified by the *rank* conjunction ⍤, match treats the entire argument as a single cell and returns a single `1` or `0`. Modifying ≡ with the rank conjunction, you can define the cells to which it applies. For example:

`L ≡⍤1 ω` ⍝ Compare list `L` with each row of ω.

`T ≡⍤2 ω` ⍝ Compare table `T` with each table of ω.

Since empty arrays do not have items which can differ from one another, they match if their ranks and shapes match:

```
      ''≡⍳0
1
```

~

Not

	Monad	*Identity Element*	*Argument Rank*
~	Not	1	0 * *

The monad ~ (*not*) is defined only on *Boolean arguments*; that is, arguments whose values contain only 1 or 0. For 1 it returns 0; for 0 it returns 1. The dyadic case of ~ is undefined.

∧ ∨ ⍲ ⍱

LCM, GCD; And, Or; Nand, Nor

	Numeric Arguments	*Boolean Arguments*	*Identity Element*	*Argument Rank*
∧	LCM	And	1	* 0 0
∨	GCD	Or	0	* 0 0
⍲	[undefined]	Nand	[none]	* 0 0
⍱	[undefined]	Nor	[none]	* 0 0

Boolean Arguments

For Boolean arguments (0 and 1), the ∨ is equivalent to the logical *or*, and the ∧ to logical *and*. Thus:

```
      p←0 1
      p ∘.∧ p
0 0
0 1

      p ∘.∨ p
0 1
1 1
```

The verbs ⍲ and ⍱ (called *nand* and *nor*) are defined only on Boolean arguments. They are negations of ∧ and ∨; that is:

$$\alpha \barwedge \omega \iff \sim\alpha\wedge\omega$$

$$\alpha \barvee \omega \iff \sim\alpha\vee\omega$$

```
      p←0 1
      p ∘.⍲ p
1 0
0 0

      p ∘.∧ p
1 1
1 0
```

Non-Boolean Arguments

For non-Boolean numeric arguments, the expression $\alpha \wedge \omega$ returns the least common multiple (LCM) of α and ω, while $\alpha \vee \omega$ returns the greatest common divisor (GCD). For example:

```
      12∨30
6
      12∧30
60

      12 30÷12∨30
2 5
      ∨/12 30÷12∨30
1

      3.6∨4
0.4
      3.6∧4
36

      x ← 6250 8750 11250 16250
      x ÷ ∨/x
5 7 9 13

      y ← 0.03135 0.04389 0.05643 0.08151
      y ÷ ∨/y
5 7 9 13

      Fraction ← 17÷64
      (Fraction,1) ÷ ∨/Fraction,1
17 64
```

Except when one of the arguments is 0, the GCD and LCM are related by a duality akin to DeMorgan's law for Boolean ∧ and ∨:

$$\alpha \vee \omega \iff \alpha \; \wedge^{\cdot\cdot} \div \; \omega$$

When one argument is 0, the result of LCM is `0` and the result of GCD is the other argument.

Tolerant GCD and LCM

For arguments other than Booleans, $\alpha\wedge\omega$ and $\alpha\vee\omega$ are subject to the visible value of `⎕ct`, comparison tolerance. Tolerant computation permits reasonable results with fractions that cannot be precisely represented. For example, the least common multiple of `0.6` and `1.4` is `4.2`. This result is returned correctly with the default value of `⎕ct`. But if `⎕ct` is set to `0`, SHARP APL fails to detect that each is an integer multiple of `0.2`, and hence returns an LCM as if they had no common factor:

```
      ⎕ct
1.419697693e¯14
      .6 ∧ 1.4
4.2
      ⎕ct←0
      .6 ∧ 1.4
3.02641895e16
```

⊢ ⊣
Right, Left; Dex, Lev

	Monad	***Dyad***	*Identity Element*	*Argument Rank*
⊢	Right	Right; Dex	[none]	∞ ∞ ∞
⊣	Left	Left; Lev	[none]	∞ ∞ ∞

Dyads ⊢ and ⊣

The dyads ⊢ and ⊣ return one argument and ignore the other. The argument returned is the one to which the horizontal part of the symbol points:

$\alpha\vdash\omega \iff \omega$

$\alpha\dashv\omega \iff \alpha$

The same principle applies to the monads. However, in that case there is no left argument. The definition becomes:

$\vdash\omega \iff \omega$

$\dashv\omega \iff$ [No result]

Dex is used primarily to pass an argument cell into a composition without changing it. For example, the *cut* conjunction modifies a monadic verb to apply to each segment of an array argument. If you wish merely to select the segments without transforming them, ⊢ does it.

Left, or *lev*, is used primarily to assign a value that cannot otherwise be syntactically fitted into an expression. For example, mathematicians are fond of sentences in the form "***x*** is such-and-such a function of ***y***, where ***y*** is ..." That is really two sentences, linked by "where." In "A Dictionary of APL," Iverson illustrates a similar construction thus: *The following expression determines the coefficients* `c` *of a polynomial equivalent to a polynomial with roots* `r`*:*

```
c←s+.×p←r×.*t ⊣s←(⍳1+n)∘.=+⌿t ⊣t←b⊤⍳×/b ⊣b←n⍴2 ⊣n←⍴r
```

In this style, you start with the conclusion `c←s+.×p` and then fill in the subordinate definitions. If this were broken into a succession of short sentences, they would appear like this:

```
n←⍴r
b←n⍴2
t←b⊤⍳×/b
s←(⍳1+n)∘.=+⌿t
p←r×.*t
c←s+.×p
```

In the foregoing, each of the segments linked by ⊣ assigns a value to a noun used in the root phrase. If there were no assignment, whatever is to the right of ⊣ would be completely discarded during evaluation. This might be the desired outcome when the material to the right is a comment, intended for the human reader but to play no part in the interpreter's execution. For example, in the following, the text that appears as the left argument of ⊢ is discarded, while the value to the right is passed through:

```
a←⍵× 'count to n'⊢⍳n ←'length of r'⊢⍴r.
```

The verbs ⊣ and ⊢ are regular in their syntax and provide an alternative to the irregular separators ◇ and ⍝.

⊤ ⊥

Encode; Decode

	Monad	***Dyad***	*Identity Element*	*Argument Rank*
⊤	[Undefined]	Encode; Represent	[none]	* ∞ ∞
⊥	[Undefined]	Decode; Base	[none]	* ∞ ∞

Dyad ⊤

The dyad ⊤ constructs the *representation* of each number in ω in the number system whose *base* (also called *radix*) is α. The dyad ⊥ is the inverse of encode, the *base value* of the representation ω in that number system.

For example, to represent a time interval recorded as 3723 seconds as a triplet in base α←24 60 60 (hours, minutes, seconds):

```
      24 60 60⊤3723
1 2 3
```

And conversely:

```
      24 60 60⊥1 2 3
3723
```

Relation Between Representation and Residue

A representation is calculated as a series of residues. This is made explicit in the following definition, which calculates a representation of ω in base α, when ω is an item and α is a list:

```
    ∇ z←α rep ω; pv; i
[1]   i←⎕io+⍴z←α,α
[2]   pv←weights α
[3]  l0:→(⎕io>i←i-1)↑0
[4]   z[i]←α[i]|ω÷pv[i]
[5]   ω←ω-z[i]×pv[i]
[6]  →l0
    ∇
```

In the definition, *pv* is the *position value* for the base. For example, in the familiar decimal system of representation, the right-most position is the units position, the next tens, the next hundreds, and so on. A length-4 representation in base 10 has position values 1000 100 10 1. The value of *pv* is calculated by the verb *weights*, defined thus:

```
      weights: 1↓⌽×\⌽α,1
```

No matter what the value of α, the value of the right-most position returned by *weights* is always 1.

Note that ⊤ is *not* tolerant (that is, is not sensitive to □*ct*).

There is a limit to the range of values that can be represented in a base of a given length. The largest integer representable in base α is α⊥α−1. More generally:[15]

```
      α⊥α⊤ω  ⟺  (×/α)|ω
```

rather than ω; for example:

```
      (3⍴10)⊤3247
2 4 7
      10⊥(3⍴10)⊤3247
247
```

The result 247 is (×/3⍴10)|3247.

However, as a consequence of the definition of 0-residue, when the base begins with 0, there is no limit to the value of the first element of the representation and any necessary extra can be accumulated there:

```
      0 10 10⊤3247
32 4 7
```

Since the last position in any representation is the *units* position, the fractional part of ω always appears in the last position of the result:[16]

```
      (3⍴10)⊤247.33
2 4 7.33
```

Combining the previous two remarks, the 0 1 representation of a number separates its integer and fractional parts:

```
      0 1⊤247.33
247 0.33
```

When a negative number is represented with a positive base, because the representation is derived from the ×/α residue, the value returned is the *complement* in that base. For example:

```
      (4⍴10)⊤¯1
9 9 9 9          Tens complement of ¯1
```

[15] With □*ct*←0 so that | is not tolerant.

[16] Fractional values may also occur in other parts of the representation when the base itself contains fractions.

```
      (4⍴16)⊤¯1
15 15 15 15
```

Familiar (to those who use base-16) as *FFFF*

```
      0 1⊤¯247.33
¯248 0.67
```

Higher-Rank Arguments to ⊤

Fundamentally, *encode* takes a number from ⍵ and represents it according to a base specified in ⍺. The number being encoded is an item. The base is specified by a list.

When ⍵ contains many numbers, the result contains a representation for each of them. In the result, the representations are spread along the *first* axis;[17] for example:

```
      (4⍴10) ⊤ 1987 2001 ¯44
1 2 9
9 0 9
8 0 5
7 1 6
```

When ⍺ contains many bases, the result contains a representation for *each* number in *each* base.

Within ⍺, the list that defines each base is spread along the *first* axis of ⍺. For example, suppose you want numbers to be represented with base `10 10 10 10` and also with base `0 1000 1000 1000`. The two bases must form a 4-by-2 array thus:

```
10    0
10 1000
10 1000
10 1000
```

Similarly, when ⍺ has shape `3 4 5`, dyad ⊤ treats it as a 4-by-5 table of three-element bases.

In the result produced by ⊤, each representation is spread along the *first* axis. When there are several representations for each number in ⍵, you get first the axes corresponding to the various representations, then the axes of ⍵. Thus:

```
⍴⍺⊤⍵  ⟷  (⍴⍺),⍴⍵
```

[17] This use of the first axis is inconsistent with the general rule that an argument cell occupies the *last* axes. The present convention was adopted in 1967 for the sake of parallelism between ⊥ and +.×. Representations along the last axis are achieved by conjoining represent with rank: `⍺⊤⍤1 0 ⍵`; see "Forcing Representation Cells to the Last Axis", below.

For example, suppose *a* contains the two bases:

```
      a←(4⍴10), ⍪0 1000 1000 1000
```

Then, the two representations of a single items are returned as a 4-by-2 table:

```
      a⊤1987
1     0
9     0
8     1
7   987
```

And the two representations of three items are returned as a 4-by-2-by-3 array, thus:

```
      a ⊤ 1987 2001 ¯44
  1   2    9
  0   0   ¯1

  9   0    9
  0   0  999

  8   0    5
  1   2  999

  7   1    6
987   1  956
```

Forcing Representation Cells to the Last Axis

You can obtain a more consistent placement of the new axis (that is, putting it at the end) by using the rank conjunction ⍤ to specify explicitly the cells of each argument. When you do that, the standard rules for frame-building apply. For example, the following expression generates all the distinct representations possible in three binary digits and makes each representation a row of the resulting table:

```
      2 2 2⊤⍤1 0 (⍳2*3)-⎕io
0 0 0
0 0 1
0 1 0
0 1 1
1 0 0
1 0 1
1 1 0
1 1 1
```

By electing to modify ⊤ with the rank conjunction, you give up its special rules for higher-rank arrays and instead apply the general rules for agreement of frames: either the frames must match, or, when one frame is empty, its single cell is paired with every cell in the other frame.

Dyad ⊥

Dyad ⊥ is inverse to dyad ⊤. While ⊤ generates the representation of a number in a particular base, ⊥ returns the numeric value of a representation ⍵, interpreted in base ⍺.

As with ⊤, the base is described by a list. However, the base may also be an item, in which case SHARP APL treats it as a list having the same length as the first axis of ⍵.

As with ⊤, for each base there is a corresponding set of *position values*; they can be calculated by the same verb `weights` described in the preceding section, "Relation Between Representation and Residue". The base is `10 10 10 10`, the position values are `1000 100 10 1`.

The base-value of a representation is the sum of the numbers in the representation, weighted by their position values:

```
⍺⊥⍵ ⟺ +/⍵×weights ⍺
```

For example:

```
      ⍺←24 60 60
      ⍵←1 2 3
      ⍺⊥⍵
3723

      weights ⍺
3600 60 1

      +/⍵×weights ⍺
3723
```

Item Left Argument

When ⍺ is an item, it is treated as `(¯1↑⍴⍵)⍴⍺`. Thus:

```
      2⊥1 0 1
5
```

This permissive treatment of the left argument is possible for ⊥ but not for ⊤.

Higher-Rank Right Argument

Fundamentally, *base* takes a number-representation from ⍵ and evaluates it according to a base specified in ⍺. The representation being evaluated is a list, and the base is a list of the same length.

When ⍵ contains many representations, the representations are assumed to be spread along the *first* axis. For example, suppose *b* is the following 4-by-3 table:

```
      b
1  2  9
9  0  9
8  0  5
7  1  6
```

The base-10 values of its three representations are:

```
      (4⍴10) ⊥ b
1987 2001 9956
```

When you provide a right argument of higher rank, the representations are still assumed to be spread along the first axis. For example, if ⍵ is 4-by-5-by-6 array, SHARP APL treats it as a 5-by-6 array of four-item representations.

Higher-Rank Left Argument

The base is assumed to be spread along the *last* axis of ⍺. For example, if you provide a 2-by-3-by-4 left argument, SHARP APL treats that as a 2-by-3 array containing six bases, each of four elements.

The result contains a base-value in *each* base, for *each* representation.

The shape-agreement rules for ⊥ are the same as those for inner product. The last axis of ⍺ must match the first axis of ⍵. This axis disappears from the result (since the value is the weighted sum along it).

The remaining axes are reproduced in the result. The shape of the result is:

```
⍴ ⍺⊥⍵  ⟺  (¯1↓⍴⍺), (1↓⍴⍵)
```

Length-1 Base

As noted, when the left argument of ⊥ is an item, it is treated as a list whose length matches the length of the representations in ⍵. More generally, whenever the last axis of ⍺ has length 1, it is assumed to apply to the representation in ⍵, and so is extended to match the length of the first axis of ⍺.

Forcing Representation Cells to the Last Axis

Just as you can use the rank conjunction to force the representation to lie along the last axis of the result of ⊤, so you can force ⊥ to accept arguments in which the base is specified by the *last* axis of ⍺, and representation cells are found along the *last* axis of ⍵.

Suppose *b* is a table containing two bases, `24 60 60` and `3⍴100`, thus:

```
      b
 24  60  60
100 100 100
```

Then you can evaluate the representation `1 2 3` in each base by:

```
      b ⊥⍤1 ⊢1 2 3
3723 10203
```

Or suppose *c* is the following four-column table:

```
      c
1 0 0 0
0 1 0 0
0 0 1 0
```

You can treat *c* as three representations, each of length 4, and evaluate each of them in base `0 24 60 60`, thus:

```
      0 24 60 60 ⊥⍤1 c
86400 3600 60
```

(This gives you the number of seconds in a day, an hour, and a minute.)

By electing to modify ⊥ with the rank conjunction, you give up its special rules for higher-rank arrays and instead apply the general rules for agreement of frames: either the frames must match, or the single cell in an empty frame is paired with every cell in the other frame.

Limitations on Inverse

In principle, ⊤ and ⊥ are inverse to each other. But this is restricted in two ways:

- ***Base.*** The representation returned by ⍺⊤⍵ is in fact the representation of `(×/⍺)|⍵`. When ⍵ is outside the range `0` through `¯1+×/⍺`, information is lost, and ⍺⊤⍺⊥⍵ is not equal to ⍵.
- ***Higher-rank arguments.*** When ⍺ is a table of bases, ⊤ and ⊥ have different assumptions about how they are arranged and how they affect the result. The inverse relationship between ⊤ and ⊥ holds only when

⍺ is a list, or when both ⊤ and ⊥ are modified by the rank conjunction, as `⊤⍤1 0` and `⊥⍤1`.

Evaluation of Polynomials

If `c` is a table containing the coefficients of the successive terms of a polynomial, then `x⊥c` evaluates the polynomial. The table of coefficients must contain a position for each term of the polynomial, in descending order; that is, starting with the coefficient for the x^n term and down, in order, to the constant (that is, the coefficient of x^0). This is the common order for the representation of numbers; however, polynomials are often written in the reverse order, with the low-order term first.

⍕

Format

	Monad	***Dyad***	*Identity Element*	*Argument Rank*
⍕	Format	Format	[none]	∞ * *

The *format* primitive ⍕ accepts any array as argument and produces a character representation of that array as a result.[18]

The result of ⍕ has the same rank as the argument, except in cases of formatting boxed arrays or non-character scalars. However, while axes other than the last have the same length as the argument, the last axis is often longer. Each numeric item in the argument is represented in the result by a format that usually requires several characters, perhaps including blanks as delimiters.

The representation of numbers is affected by the following system nouns:

`⎕pp` ***Print precision.*** The number of significant digits used by *monad* ⍕.

`⎕ps` ***Position and spacing.*** Alignment within and between cells for display of an array of boxes by *monad* ⍕.

`⎕fc` ***Format control.*** Format conventions for *dyad* ⍕.

Default Display

SHARP APL automatically displays the final result of any sentence that has a final result but does *not* assign that result to a name. Similarly, any noun

18 The system verb `⎕fmt` provides a wider variety of formats than those available with ⍕, controlled in a somewhat different fashion. It is described in Chapter 16, "System Nouns and Verbs".

assigned to □ or ⍞ is displayed. In these cases, where format need not (and cannot) be explicitly specified, the format is the same as that produced by monad ⍕.

Monad ⍕

Monadic *format* converts an array whose items are numbers or boxes to a character representation. Packages are outside its domain. Applied to an array of characters, ⍕⍵ returns ⍵ unchanged; that is, ⍵≡⍕⍵.

Applied to an array of boxes, ⍕⍵ opens and formats the contents of each box independently, and then arranges the separate item-formats into a grid representing the structure of the array (as is explained later in this section).

Applied to a numeric array, ⍕⍵ returns a character array in which the numeric values are represented in one of three possible formats:

- ***Fixed-point.*** This form shows a leading zero only in the units position and does not print trailing zeros to the right of the decimal point. The decimal point appears only when there is a digit to the right of it (so *integer format* is simply a special case of fixed-point format). A negative number has a macron (high minus sign) before the first digit. For example:

```
      ⍕7 8 12 ÷ ¯8 8 ¯8
¯0.875 1 ¯1.5
```

- ***Exponential.*** This form, also called *scientific notation* or *engineering format*, consists of two parts, *mantissa* and *exponent*, joined by the letter *e*. In scientific notation, the mantissa is a number less than 10 and at least 1 (except when the number being represented is 0), written in fixed-point format. The exponent is an integer indicating the power of ten by which the mantissa must be multiplied. For example:

```
      ⍕125 × 10*12
1.25e14
      ⍕¯7 * 12 13
1.38412872e10 ¯9.688901041e10
```

  Engineering format is similar to scientific notation, except that the exponent is always a multiple of three.

- ***Complex.*** This form consists of the real and imaginary parts linked by the letter *j*. The parts may be represented in any of the formats (integer, fixed-point, or exponential), independently of each other. For example:

```
      ⍕27 + 0j1e12
27j1e12
```

When the imaginary part is 0, monad ⍕ includes neither the character `0` nor the letter *j* (and the format of the real part is the same as fixed-point or exponential format). For example:

```
      ⍕¯50.36j47.52 + 3j4 4j5 × 0.12j¯16
14 30.12j¯15.88
```

Array-wide Format vs. Item-by-Item Format

Under some circumstances, a single format is adopted for an array as a whole, and all items are formatted and aligned in the same way. Under other circumstances, each item in an array is formatted independently.

An array-wide common format is adopted when:

- the array's rank is greater than 1

  and

- no item in the array uses *j* format (that is, has a non-zero complex part).

Conversely, independent item-by-item formatting is adopted when:

- the array has rank 1 or less

  or

- any item in the array uses *j* format (that is, has a non-zero complex part).

When a common format is adopted for an entire array, either all items are shown in fixed-point format, or all items are shown in exponential format. In either event, all columns have a common *field width.*[19] Within each field, the units position of each representation is at the same place, so that it is vertically aligned with the others.[20] In exponential format, the units positions of the mantissa are aligned.

Choice of Fixed-Point or Exponential Format for Real Numbers

When ⍵ is real[21] SHARP APL selects a format as follows:

- A number[22] whose magnitude lies in the range `10*¯6` to `10*⎕pp` is represented in *fixed-point* format.
- A number that is exactly equal to zero is represented as `0`.
- A number outside those ranges is represented in *exponential format.*

19 See the discussion of "Field Width in Tables", later in this section.

20 When there are decimal points, they are aligned too. But since the point is omitted when there are no digits after it, it is safer to say that the units positions are aligned.

21 "Real" is used here in the mathematical sense of "having no imaginary part"; it does not exclude integers.

22 Or part (real or imaginary) of a complex number.

Notice two useful consequences of these rules:

- In fixed-point format, no value of ⎕*pp* can discard digits to the left of the decimal point.
- A number for which SHARP APL's rules require exponential format could not be written more briefly in fixed-point format.

In an array having two or more axes and in which no element requires a *j* in its representation, SHARP APL uses *exponential* format for the entire array when *any* of the following is true:

- The largest magnitude in the array is greater than `10*⎕pp`.
- The smallest magnitude in the array is less that `10*-⎕pp+4`.
- The range from the smallest to the largest magnitude exceeds a factor of `10*⎕pp+4`.

Mixed Formats in Representation of a List

In a list, SHARP APL selects fixed-point or exponential format for each of the items independently. For example:

```
      ⎕pp←5
      ⍕0.2×10*6×¯3+⍳6
2e¯13 0.0000002 0.2 2e5 2e11 2e17
```

Appearance of Fixed-Point Format

A number formatted by monad ⍕ in fixed-point format is displayed with:

- no more than ⎕*pp* significant digits.
- no trailing zeros to the right of the decimal point.
- A leading zero to the left of the decimal point *only* in the units position.
- A decimal point only when there is a digit to the right of it.
- A macron (high minus) ¯ before the first digit of a negative number.

For example:

```
      ⎕pp←4
      ⍕⍪ - 10000 100 1÷3
¯3333
  ¯33.33
   ¯0.3333
```

Appearance of Exponential Format

In *exponential* format, each value being formatted is represented by a pair of numbers, the *mantissa* and the *exponent*, with an *e* between them.

The exact form of the mantissa depends on whether each item is formatted independently or there is an array-wide uniform format. When items are formatted independently, trailing zeros to the right of the decimal point are omitted, and the point is included only when there are digits to the right of it. For example:

```
      ⍕10 11 12 * 10 ⊣ ⎕pp←4
1e10 2.594e10 6.192e10
```

However, with an array-wide uniform format, every item has the same mantissa format and the same exponent format. If any mantissa requires a decimal point, they all have one. If any requires positions to the right of the decimal point, they all do. The greatest number of trailing decimals is ⎕*pp*−1.

The exponent is expressed as an integer. It may have one or two digits and perhaps a negative sign (when the magnitude being represented is less than 1). Space is allocated to accommodate the exponents needed for the actual data; at most, the exponent takes three print positions (to accommodate a negative two-digit exponent). For example:

```
      ⍕3 2⍴10*2/5 6 7
1e5 1e5              There is only one digit before e
1e6 1e6              because all mantissas can be represented
1e7 1e7              by a single digit.

      ⍕3 2⍴10*5 5 6 6 7 70
1e5  1e5             Both columns have space for a two-digit
1e6  1e6             exponent even though only one column
1e7  1e70            requires two digits.

      ⍕¯1 1 ×⍤1 ⊢3 2⍴10*5 5 6 6 7 70
¯1e5   1e5           Both columns have two positions before e
¯1e6   1e6           even though only one column uses both
¯1e7   1e70          of them.

      ⍕(10*5 6 7), 3 1⍴10 11 12 * 10 ⊣ ⎕pp←4
1.000e5  1.000e10    The mantissas on the left have trailing
1.000e6  2.594e10    zeros to make them match the width
1.000e7  6.192e10    of mantissas on the right.
```

The exponent is always positioned flush-left next to the *e*. Thus, even when the letter *e* is aligned, the units position of the exponents is *not* necessarily aligned. For example:

```
      ⎕pp←4
      ⍕⍪0 2, (÷3)×10*10 11 ¯12
0.000e0
2.000e0
3.333e9
3.333e10
3.333e¯13
```

Appearance of Complex Format

When any item in an array has a non-zero imaginary part, that item is displayed in complex format, with a *j* between the real and imaginary parts. (Conversely, in a complex array, an item whose complex part is zero is displayed in the usual fixed-point or integer form.)

The real and imaginary parts of each number are formatted independently; each may be in fixed-point format or in exponential format, without regard to the format of the other. There is no array-wide uniform format for an array in which any item is represented in complex format.

In a list, the representations of complex numbers may vary in length. Successive items are separated by one blank, thus:

```
      ⍕(- 2 4 6 8) *÷3 ⊣⎕pp←4
0.63j1.091 0.7937j1.375 0.9086j1.574 1j1.732
```

For an array of rank 2 or higher, each number's representation appears flush-right. Within the field devoted to a column of ω, there is no attempt to align the letter *j* or either of the units positions.

```
      ⍕⍪(- 6 8 0 66 68)*÷3 ⊣⎕pp←4
0.9086j1.574
     1j1.732
           0
   2.021j3.5
 2.041j3.535
```

Field Width in Tables

Field width is the number of character-positions allocated to receive the representation of a number. When SHARP APL formats a table, it calculates a single value of field width for the entire table. The field width and

the location of the units position within it are calculated as though all the numbers were in a single column.

The field width adopted throughout an array is the minimum width in which all the values can be represented. The field includes an extra column for the negative sign only when needed to represent a value whose representation extends all the way to the left side of the field.

The maximum field width of an array in either fixed-point or exponential format is 6+⎕*pp*.[23] Thus the rule for selecting one format or the other can be restated thus: fixed-point format is used *unless* it would require a width greater than 6+⎕*pp*.

The field width of a complex array of rank 2 or greater is the length of the longest representation. The greatest possible width is thus 1 (for the letter *j*) plus 2×6+⎕*pp*.

Blanks Between Columns

In every array having two or more axes and two or more columns, one column of blanks is inserted between adjacent fields.

Examples of Monad ⍕

The following examples are intended to illustrate the points just mentioned. In particular, you can see that the same format is used throughout the display of a real numeric table:

```
      ⍕2 4⍴÷1 2 1 2
1   0.5 1   0.5
1   0.5 1   0.5

      ⍕2 4⍴÷3 1 2 3
0.3333333333 1            0.5          0.3333333333
0.3333333333 1            0.5          0.3333333333

      ⍕¯1 1 ∘.× 2 4⍴÷3 1 2 3⊢⎕pp←5
¯0.33333 ¯1       ¯0.5     ¯0.33333
¯0.33333 ¯1       ¯0.5     ¯0.33333

 0.33333  1        0.5      0.33333
 0.33333  1        0.5      0.33333
```

[23] For the exponential format, that is ⎕*pp* digits, plus the decimal point, the letter *e*, possibly a sign for the value and another for its exponent, and possibly two digits for the magnitude of the exponent.

Separation Between Planes of Higher-Rank Arrays

As noted at the beginning of this section, in general the result of ⍕ has the same rank as its argument. In displaying any array of rank greater than 2, SHARP APL inserts an extra blank line to separate successive rank-2 segments (tables), two extra blank lines to separate successive rank-3 segments, and so on. You can see such a blank line in the preceding example.

Neither the *newline* character at the end of each row, nor the *newline* characters required to produce these additional separations, are part of the result of ⍕, but appear as a consequence of the number of axes in the result.[24]

Format of an Array of Boxes

When ⍵ is an array of boxes, ⍕⍵ is always a table. Each of the boxes is opened and formatted independently, without regard to the formatting of any of the other boxes. The display of each box looks the way it would if the object within were to be displayed alone. The display for each box is a table, regardless of the rank of the object being displayed. When the array enclosed in the box has rank greater than 2, the separations between sub-arrays appears as extra rows of blanks.

The process of opening boxes and displaying their contents is recursive. That is, a box may itself contain boxes, and when it does, each is formatted according to the same rules.

When a table has been computed for each box independently, they are assembled into a grid corresponding to the structure of the array. However, the tables that represent the various boxes may vary in size. Each column (that is, each position along the *last axis*) is allocated a common width sufficient to accommodate the widest table in that column. Similarly, each row (that is, each position along the *next-to-last* axis) is allocated a common height sufficient to accommodate the tallest table in that row. This preserves the alignment of the boxes, albeit by creating a grid whose rows and columns vary in width. The effect – rather like a plaid blanket – is illustrated in Figure 6–10.

[24] However, when monad ⍕ is applied to an array of boxes, the resulting representation never has rank greater than 2 and in that situation higher-rank arrays are indeed depicted by inserting additional blank rows.

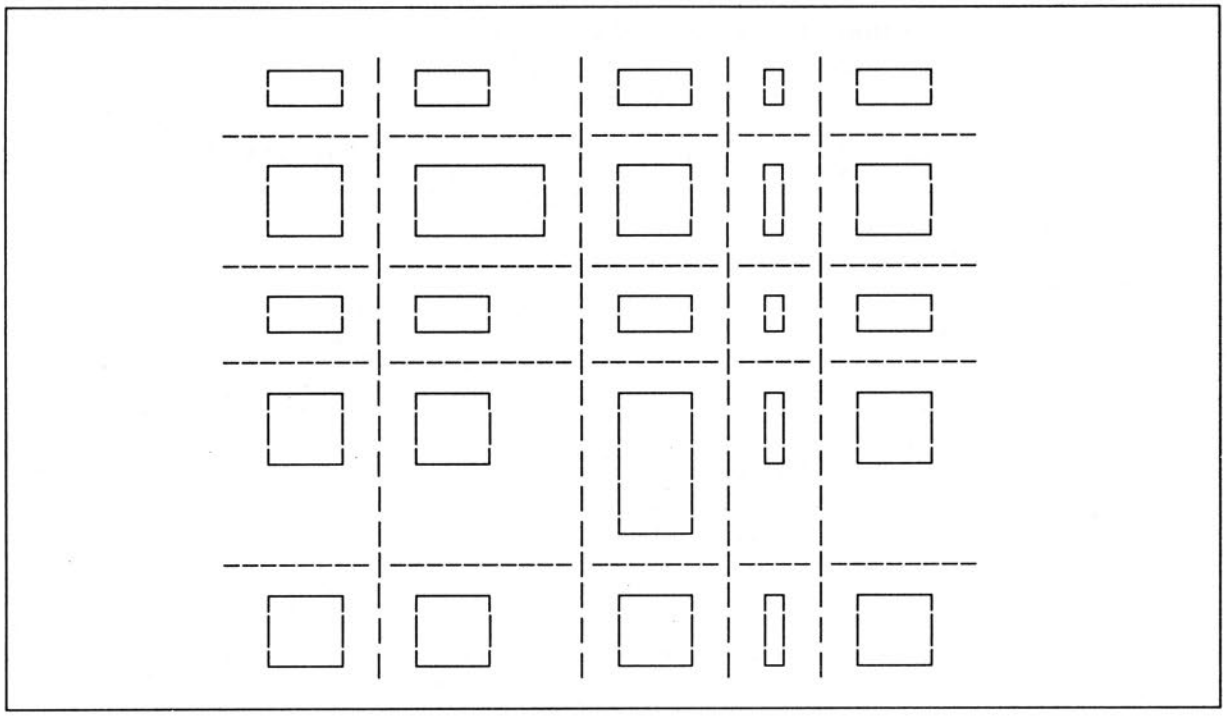


Figure 6–10 : Preservation of grid structure in formatting an array of boxes.

In Figure 6–10, where the grid provides a box more space than it needs, each box appears at the top left corner of its space.

Effect of □*ps* on Position and Spacing of Boxes

Each box is allocated height and width sufficient to accommodate the tallest box in its row or the widest box in its column. Unless the boxes have displays of the same shape and size, some will have more space allocated than they need. How boxes are positioned within the space available to them is controlled by the first two elements of the system noun □*ps*. Vertically, boxes may be at the top, center, or bottom of the available space; horizontally, they may be at the left, center, or right. This control applies to the array as a whole; there is no direct provision to position some boxes one way and some another. The default value of □*ps* is `¯1 ¯1 0 1`.

To separate adjacent boxes, you can set □*ps*`[3 4]` so that additional blanks are inserted between the rows and columns. Provided □*ps*`[3]` and □*ps*`[4]` have a magnitude no less than 2, you may also elect to have horizontal or vertical demarcation lines around each box.

Position: The First Two Elements of ⎕*ps*

⎕*ps*[1]	¯1	At the top of the rectangle
	0	Vertically centered in the rectangle
	1	At the bottom of the rectangle
⎕*ps*[2]	¯1	At the left of the rectangle
	0	Horizontally centered in the rectangle
	1	At the right of the rectangle

Spacing: The Last Two Elements of ⎕*ps*

⎕*ps*[3]	Magnitude	Number of positions separating vertically adjacent items
	When negative	Draw lines at top and bottom edges of boxes if space permits
⎕*ps*[4]	Magnitude	Number of positions separating horizontally adjacent items
	When negative	Draw lines at left and right edges of boxes if space permits

When ⎕*ps*[3] or ⎕*ps*[4] is negative, the character ¯ is used to mark the top of a box, the character _ to mark the bottom, and the character | to mark the sides. Where a box touches the edge of the space assigned to it, the border characters surrounding the box overwrite the first of the separating blanks on that edge. For that reason, border characters may be drawn around a box only when there is space for them: that is, when the magnitude of the controlling item in ⎕*ps* is at least 2.

At the edge of the result of ⍪⍵ there are no separating blanks. There, the border characters are placed in an additional row or column. Thus, when ⎕*ps*[3] is ¯2 or less, the result has two extra rows, one at the top and one at the bottom. Similarly, when ⎕*ps*[4] is ¯2 or less, the result has two extra columns, one at the left and one at the right edge.

The following example illustrates the effects of ⎕*ps* on an array of boxes:

```
      B←2 2⍴(3 5⍴'AB')⊃1 2⊃'CD'⊃'EFGHIJ'
      ⍒B ⊣⎕ps←¯1 ¯1 ¯3 ¯3 ⍝ Top left, draw boxes
|¯¯¯¯¯| |¯¯¯|
|ABABA| |1 2|
|BABAB| |___|
|ABABA|
|_____|

|¯¯|    |¯¯¯¯¯¯|
|CD|    |EFGHIJ|
|__|    |______|

      ⍒B ⊣⎕ps←¯1 ¯1 0 0    ⍝ Top left, no spacing
ABABA1 2
BABAB
ABABA
CD    EFGHIJ

      ⍒B ⊣⎕ps←0 0 0 0      ⍝ Centered, no spacing
ABABA
BABAB 1 2
ABABA
 CD   EFGHIJ
```

Dyad ⍒

The result of $\alpha⍒\omega$ is a character array in the same way as the result of monad ⍒. The left argument permits more specific control of format and field width. It describes a series of *fields*, each to contain the representation of a *column* of ω. The fields are described in three different ways, depending upon whether α is *empty*, *numeric*, or *character*.

Empty α The right argument is formatted according to the rules for monadic format, as if the expression were $⍒\omega$ instead of $\alpha⍒\omega$.

Numeric α Each field is described by a pair of integers. In general, α is a list with twice as many items as ω has columns. The first member of each pair sets the width of the corresponding field. The second member of each pair sets the number of fractional positions or the number of significant digits.

Character α "Format by example." α is a character list having the same length as the last axis of the result. Embedded within α are characters indicating by example the position and format of the various fields.

Dyad ⍕ with Empty α

When α is an empty vector, α⍕ω formats ω according to the rules for monadic format, as if the expression were ⍕ω. This is useful when creating reports containing mixed character and numeric data:

```
      name←4 5⍴'MariaPhil ElenaKen  '
      pay←4 1⍴ 34000 125000 193000 17000
      id←⍪123 456 789 666
      t←1 3⍴'Employee name  '⊃'Userid  '⊃'Salary '
      ⍕t,⍪(''⊃5 0⊃6 0)⍕⍤1 0⊃name⊃id⊃pay ⊢ ⎕ps←0
Employee Name  Userid  Salary
     Maria        123   34000
     Phil         456  125000
     Elena        789  193000
     Ken          666   17000
```

Dyad ⍕ with Numeric α

When ω is numeric, α⍕ω formats each of its columns according to the specifications encoded in α. As with monad ⍕, "column" here means a position along the last axis.

The left argument α requires two integers to specify the format of a column of ω. Each pair of integers in α controls the formatting of a column of ω. Alternatively, α may be a single pair, which applies to *all* the columns of ω, or a single integer, in which case it is treated as if it were the pair ¯2↑α. Otherwise, `length error` occurs.

Within each α-pair, the first integer controls the *width* of the resulting representation. The second integer controls the *type* of representation (fixed-point or exponential), and the *number of places* to the right of the decimal point (in a fixed-point representation) or the *number of significant digits* (in an exponential representation).

Both types of representation, fixed-point and exponential, are *right-justified* within their fields. In fixed-point representation, the number of places beyond the decimal point is fixed and filled with trailing zeros.

To provide blanks for visual separation of fields, assign a width greater than needed to represent the values in ω, so that the field has leading blanks.

Fixed-Point Representation

The first member of an α-pair specifies the *total width* of the field. However, when the first member of the field is 0, the field is made wide enough to

accommodate the widest representation in that column, plus one leading blank.

The second member of an α-pair specifies the *number of places to the right of the decimal point.* When the number of places is greater than `0`, an *additional position* is required for the decimal point itself. Unused positions to the right of the decimal point are filled with trailing zeros.

To represent a negative value in ω, the negative sign `¯` is inserted to the left of the left-most digit. When calculating the width required for a field, you should allow space for the negative values actually present in the data. When the field lacks space to print the sign for a negative value of ω, there is a *field overflow*. What SHARP APL does then depends on the value of `⎕fc[4]`. (See the discussion of `⎕fc` later in this section or in Chapter 16, "System Nouns and Verbs".)

To illustrate formatting rules, consider the table *m* created as follows:[25]

```
      m←((÷3) × 10*⍳5) ∘.× 1 1 ¯1 ⊣⎕pp←5
      ⍕m
    3.3333      3.3333     ¯3.3333
   33.333      33.333     ¯33.333
  333.33      333.33     ¯333.33
 3333.3      3333.3     ¯3333.3
33333       33333      ¯33333
```

To show the boundaries of the result, catenate brackets to the left and right:

```
      '[',(8 2⍕m),']'
[    3.33    3.33   ¯3.33]
[   33.33   33.33  ¯33.33]
[  333.33  333.33 ¯333.33]
[ 3333.33 3333.33¯3333.33]
[33333.3333333.33********]
```

At the lower left, representations run together because a field width of `8` is insufficient to leave an initial blank. At the lower right, there is field overflow because field width `8` is insufficient to represent the number `¯33333.33`.

Field width `0` causes `⍕` to calculate the width of the fields. A single pair beginning with `0` computes the minimum common field width for the entire array:

25 Recall that `⎕pp` affects monad `⍕` but not dyad `⍕`.

```
      '[',(0 2⍕1,m),']'
[      1.00      3.33      3.33     ¯3.33]
[      1.00     33.33     33.33    ¯33.33]
[      1.00    333.33    333.33   ¯333.33]
[      1.00   3333.33   3333.33  ¯3333.33]
[      1.00  33333.33  33333.33 ¯33333.33]
```

Specifying `0 2` for each field separately permits each column to be assigned its own width, independently of the others. It allows space for a negative sign whether or not there is one in the corresponding column of ω.

```
      '[',(0 2 0 2 0 2 0 2 ⍕ 1,m),']'
[  1.00      3.33      3.33     ¯3.33]
[  1.00     33.33     33.33    ¯33.33]
[  1.00    333.33    333.33   ¯333.33]
[  1.00   3333.33   3333.33  ¯3333.33]
[  1.00  33333.33  33333.33 ¯33333.33]
```

Exponential Representation

When the second member of an α-pair is *negative*, the result is represented in *exponential form*. The first item of an α-pair sets the total width of the field, while the absolute value of the second member specifies the number of significant digits. In an exponential representation, positions are required as follows:

- The *last three* positions of each field are reserved for the exponent and its sign. They are set flush-left within those three positions; unused positions are filled with blanks.
- The *fourth* position from the end is reserved for the character *e*.
- In the remaining leftward positions, there must be space for the following:
  - The decimal point.
  - The number of digits specified in the first member of the pair. The first of these digits appears to the left of the decimal point; the others appear between the decimal point and the character *e*.
  - A negative sign, if required for the value being represented.

In general, the total width must be at least 6 greater than the number of significant digits requested. However, when the value being represented is not negative, a width 5 greater than the number of significant digits suffices.

Any unused positions at the left of the field are filled with blanks. The following illustrates some of the points mentioned:

```
      t←2/⍪ (10÷3) × 1 ¯1 1e¯30
      '[', (8 ¯3  15 ¯3 ⍕ t), ']'
[3.33e0          3.33e0  ]
[********       ¯3.33e0  ]
[3.33e¯30        3.33e¯30]
```

Dyad ⍕ with Character ⍺

When ⍺ is a character array, the expression ⍺⍕⍵ is called *format by example*: ⍺ provides a *pattern* for the result. The right argument ⍵ is a numeric array of any rank. The left argument ⍺ is a *list* of the same length as the last axis of the desired result and shows how each column of ⍵ is to be represented.

The system noun ⎕*fc* is an implicit argument to dyad ⍕ and controls how it treats certain cases, as explained below.

Numeric Fields within the Pattern

The left argument ⍺ consists of a series of numeric "fields," optionally embedded in non-numeric text. The representations of the numbers in ⍵ replace the numeric fields of the pattern. ⍺ must contain either a numeric field for *each* column of ⍵ or a *single* numeric field that is then applied to *all* columns of ⍵.

A numeric field must contain at least one digit. It may also contain the characters **.** (*dot*) and **,** (*comma*). Collectively, the digits 0 through 9 and the dot and comma are called *control characters*. (Whether a dot or comma serves as a control character depends on context, as explained below.)

A field is bounded by *blanks* or (on the right) by the first non-control character occurring further right than a 6. For example, the following has three numeric fields, pointed out by the arrows shown below. The first numeric field is ended by the colon, since colon is the first non-control character to the right of the 6.

```
'Part # 05555-60:  Stock 55550  Updated 55/55/55'
        ↑↑↑↑↑↑↑↑          ↑↑↑↑↑           ↑↑↑↑↑↑↑↑
```

Although the presence of one or more blanks serves to delimit a field, each blank is reproduced in the result.

The digits in a field reserve places in the result to receive the representation of a number. If they fail to provide sufficient positions, there is a *field overflow*. In the result, the numeric field for that value is shown only by asterisks (or by the character you indicate in ⎕*fc*[4]).

By their values, the digits in the pattern's numeric fields indicate how the number is to be formatted. Much of what follows concerns the effect of the digits `0` through `9` on the representation of a number.

The examples that follow make use of a user-defined verb *on*. It shows the left argument of ⍣ and then the result, aligned one above the other so that you can compare the result with the "picture" that controlled it. The argument of ⍣ is shown with a row of dashes before and after it, thus:

```
      a←'550 @ $340.10CR ea. = $3,540.10CR'
      d←3 2⍴10 188.4  2 ¯4  20 .12
      a on a ⍣ d,×/d
--------------------------------
550 @ $340.10CR ea. = $3,540.10CR
--------------------------------
 10 @ $188.40   ea. = $1,884.00
  2 @   $4.00CR ea. =     $8.00CR
 20 @   $0.12   ea. =     $2.40
```

Decorators within the Pattern

Characters that are not control characters are called *decorators*. In the preceding example, the characters `@`, `$`, `CR`, `ea.`, and = are decorators. So were the texts `Part #` and `Updated` in the earlier example, and the hyphen and slashes, even though they were embedded within a numeric field. In general, decorators are simply reproduced unchanged in the result. Those that are always reproduced and never float are called *simple* decorators.

A *controlled decorator* is text attached to the beginning or end of a numeric field, usually to indicate the *sign* of the number there represented. In the example, `$` and `CR` are controlled decorators because each of them abuts a numeric field.

A controlled decorator *floats*: that is, it moves inward to occupy positions that the pattern reserved for the number's representation, but that were unneeded for the value being represented. In the pattern, a controlled decorator is a sequence of one or more characters that:

- does not contain a blank
- is located immediately to the left of the numeric field's first digit, or immediately to the right of its last digit.

The way the controlled decorator text appears in the result is controlled by the presence of the digits `1`, `2`, or `3` in the numeric part of the pattern. (If the pattern does not contain one of those three digits, or there is a `4` somewhere between the decorator text and the number `1`, `2`, or `3`, text that would otherwise be a controlled decorator is treated as a simple decorator.

Treatment of the Decimal Point and Decimal Digits

In the result, the digits of a number's representation replace positions in the pattern's numeric field that contain digits. A decimal point can appear in the result *only* when it occurs as part of the pattern's numeric field. Moreover, the fractional part of a number can be represented in the result *only* when the pattern contains both a decimal point and one or more digits to the right of it.

When the pattern provides fewer positions for decimal digits than the representation requires, the representation is rounded at the right-most position provided. Thus, if you provide two decimal positions, a representation that requires more is rounded to the nearest hundredth. Similarly, if you use a pattern that makes no provision at all for decimals, the result is rounded to the nearest integer:

```
      a←'555,550 no decimals vs. 550.555'
       a on a ⍒ 2/⍪÷⍳5
------------------------------
555,550 no decimals vs. 550.555
------------------------------
      1 no decimals vs.   1
      1 no decimals vs.   0.5
      0 no decimals vs.   0.333
      0 no decimals vs.   0.25
      0 no decimals vs.   0.2
```

Treatment of the Sign

The sign is represented in the result by the *controlled decorators* in the pattern. Controlled decorator text can appear in the result in three ways: only for negative values, only for non-negative values, or for any values, regardless of sign. Controlled decorator text floats.

The following are the digits that affect controlled decorator text. (For details and examples, see the description of the various following control digits, later in this section.)

1 Include and float the controlled decorator text for a *negative* value.

2 Include and float the controlled decorator text for a *non-negative* value.

3 Include and float the controlled decorator text regardless of value.

4 Block the effect of 1, 2, or 3 when interposed between them and a decorator text.

The following illustrates a decorator to mark negative values:

```
      a←'On 05/05/05, balance was (55,510.50)'
      a on  a ⍕ 880611, ⍪1 ¯1 1 ¯1 ×2/23456.71 456.7
-----------------------------------
On 05/05/05, balance was (55,510.50)
-----------------------------------
On 88/06/11, balance was  23,456.71
On 88/06/11, balance was (23,456.71)
On 88/06/11, balance was     456.70
On 88/06/11, balance was    (456.70)
```

A negative number can be represented in the result *only* when the pattern includes a controlled decorator text *and* the numeric field includes a `1` (meaning "Include the controlled decorator when the value is negative").[26]

When your pattern fails to provide for a negative value but ⍵ contains one, you have an *overflow* condition. The numeric field for that number is replaced by asterisks.[27]

A negative value is marked in the result by whatever text you provide. There may be controlled decorator text on *both* sides of the numeric field, or only before or after. Note that if you want a leading macron as negative sign, you still have to say so explicitly; format by example does *not* have a default negative sign.

Alternative Conventions Regarding Decimal Point and Grouping

In the left argument of ⍕, only a dot can indicate where the decimal point is to be located; only a comma can indicate a grouping symbol to be inserted between digits. However, different characters may appear in the result, in accordance with national convention. In the result, the character that separates integer from fraction is `⎕fc[1]`; it replaces the dot that indicates the decimal point in the pattern. In the result, the character that separates groups of digits is `⎕fc[2]`; it replaces the commas that indicate grouping in the pattern. For example:

[26] Well, not quite only. You can represent the magnitude of a negative value when the numeric field includes a `3`, but since the controlled text is then included regardless of sign, you have lost the distinction between negative and non-negative values.

[27] Alternatively, depending on how you set `⎕fc[4]`, you may substitute some other character to indicate overflow or to report a *domain error*. See the discussion of `⎕fc` later in this section or in Chapter 16, "System Nouns and Verbs".

```
      ⎕fc[1 2]←', '
      a←'555,555,555,550.555,555,555'
      a on a⍒⍪!.5+2×⍳6
-------------------------
555,555,555,550.555,555,555
-------------------------
             3,323 350 97
            52,342 777 785
          1 871,254 305 798
        119 292,461 994 609
     11 899 423,083 962 248
  1 710 542 068,319 573 16
```

In the pattern, a dot indicates the decimal point when:

- it precedes the first digit *or* it is surrounded by digits in a numeric field

  and

- it is the only such dot in the field.

Thus, in the following pattern, the dot in `5.5` is the decimal point, but the sequence of three dots is simple decorator text (and reproduced in the result):

```
      a←'5...55.55'
      a on a⍒  4 1⍴ 1 12.3 123.45
---------
5...55.55
---------
 ... 1
 ...12.3
1...23.45
 ... 1
```

Dot and Comma

The dot and comma are *conventional* decorators; they specify decimal digits or group separators according to common conventions (as in `23,456.78`).

A comma is a grouping decorator when it is surrounded by digits in a numeric field. Thus, in the following pattern, the sequence `5,5` indicates where a grouping comma should be inserted. The other commas are not grouping commas because they are not surrounded by digits on both sides. In the result, a grouping comma appears wherever the result has a digit on both sides of it. Other commas appear according to the usual rules for

decorator text. Thus the comma in `Left,` is not a grouping comma and neither are the double commas.

The phrase `Left,` and the right-most comma in the pattern are sign decorators; because of the number 2 in the picture, they are included (and float) for non-negative values. The word `Right` is not part of the sign decorator (since there is a space between it and the rest of the picture). Thus the double comma and the word `Right` appear in every field of the result.

```
      a←'Left,55,,525,5, Right'
      a on a ⍕ ⍪12345 ¯12345 3  ¯3 123 ¯123
--------------------
Left,55,,525,5, Right
--------------------
 Left,1,,234,5, Right
      1,,234,5, Right
  Left,,,    3, Right
       ,,    3  Right
  Left,,, 12,3, Right
       ,, 12,3  Right
```

A decimal point not followed by a numerical digit is a part of the trailing decorator and as such is included or excluded in the same way as any other part of the controlled decorator text. For example:

```
      a←'-5125.POS'
      a on a ⍕⍪123 ¯123
---------
-5125.POS
---------
 123.POS
-123
```

Significance of the Control Digits 0 to 9

The control characters 0 to 9 are described here in numerical order.

0 ***Leading/trailing zeros.*** Unused positions are filled with 0 from here towards the decimal point. For example:

```
      a←'55,550.0055'
      a on  a ⍕ ⍪ 1 0 1000.1, ÷3
-----------
55,550.0055
-----------
     1.00
     0.00
 1,234.50
     0.3333
```

If you do not specify any position at which a leading or trailing zero is printed, a zero value has no printed digits. The decimal point also turns into a blank when there is no digit to the right of it.

```
      a←'[555.55]'
      a on a ⍕ 4 1 ⍴ 1 0 10.1 100
--------
[555.55]
--------
[  1   ]
[      ]
[ 10.1 ]
[100   ]
```

1 ***Negative values.*** The controlled decorator texts next to the number's representation are included and float inward until they touch the number's outermost printing digits. However, when the value being represented is not negative, the controlled decorator is replaced by blanks. For example:

```
      a←' Value (1550.50)'
      a on  a ⍕⍪1.23 ¯12.34 123.4 ¯123.4
----------------
 Value (1550.50)
----------------
 Value     1.23
 Value   (12.34)
 Value   123.40
 Value  (123.40)
```

2 ***Non-negative values.*** The controlled decorator texts next to the number's representation are included and float inward until they touch the number's outermost printing digits. However, when the number being represented is negative, the controlled decorator is replaced by blanks. For example:

```
      a←' Value +2550.50Plus'
      a on  a ⍪⍪1.23 ¯12.34 123.4 ¯123.4
------------------
 Value +2550.50Plus
------------------
 Value    +1.23Plus
 Value    12.34
 Value  +123.40Plus
 Value   123.40
```

3 ***Float.*** For all values, the controlled decorator texts next to the number's representation are included and float inward until they touch the number's outermost printing digits. (Notice that in the following example the result does not distinguish the sign of the argument.)

```
      a←'%%[35,550.55]%%'
      a on  a ⍪ 81.23 ¯2 ¯3333 .4
---------------
%%[35,550.55]%%
---------------
    %%[81.23]%%
     %%[2]%%
 %%[3,333]%%
     %%[0.4]%%
```

Two of the control characters 1, 2, and 3 may occur in the same pattern. (It is not illegal to put more than two, but since there are only two controlled decorator texts, left and right, only the outer two control characters are effective.) When there is more than a single occurrence of 1, 2, or 3, inclusion of each decorator text is governed by the control character encountered first, scanning from the decorator towards the numeric part of the pattern. For example, the pattern

```
'+215550.50Overdraft'
```

causes + to be printed to the left of a non-negative value because, scanning from the left, 2 is the first relevant control character encountered. Similarly, *Overdraft* appears to the right of a negative value because, scanning from the right, 1 is the first relevant control character.

The effects of the digits 1, 2, and 3 do *not* depend on where they are located with respect to the decimal point.

4 ***No float.*** The action of a 1, 2, or 3 is nullified by a 4 that occurs between the 1, 2, or 3 and the decorator text. If the preceding example is modified by introducing a 4 somewhere to the right of the 3, the right decorator text is now reproduced in place for all values, but the left decorator text continues to float inward, as before:

```
      a←'%%[35,550.55]%% %%[35,450.55]%%'
      a on  a ⍕ 2/¯81.23 2 3333 .4
------------------------------
%%[35,550.55]%% %%[35,540.54]%%
------------------------------
    %%[81.23]%%     %%[81.23]%%
     %%[2]%%         %%[2   ]%%
 %%[3,333]%%     %%[3,333   ]%%
     %%[0.4]%%       %%[0.4 ]%%
```

5 ***Normal digit.*** The position is available for a digit. If the representation of a particular value does not require the leading or trailing positions, the controlled decorator text moves in to occupy them (but only when a decorator text has been supplied and is controlled by the presence of a `1`, `2`, or `3` in the pattern).

6 ***Field delimiter.*** The next rightward non-control character (that is, any character other than `0123456789.,`) starts a new field. This is needed where fields are not separated by blanks. The digit `6` provides an additional field break at the next non-control character. For example, this allows you to include slashes rather than blanks as field delimiters to format three integers (for example, to represent a date):

```
      '0556/06/05' ⍕ 1988 11 4
1988/11/04
```

7 ***Exponential format.*** The value is represented in exponential notation. The next rightward non-control character (for example, *e*) separates the *mantissa* from the *exponent*. The decimal point (which must be somewhere to the left of the `7`) marks the decimal point of the mantissa.

Your pattern must provide at least one digit to the left of the point. If a value to be formatted is negative, it needs another position for the sign. By convention, in *scientific format* the mantissa has exactly *one* digit to the left of the point and the exponent is adjusted accordingly. For example, 1988 is represented as `1.988e3`. To achieve that, use `1` to mark the leading digit, with the character you are using as a negative sign to the left of it.

It is possible to specify more than one digit to the left of the point. SHARP APL selects the exponent to match the number of digits to the left of the point. For example, when you provide two digits to the left of the point, SHARP APL selects an exponent that is a multiple of 2. If your pattern has three digits to the left of the point, SHARP APL provides an exponent that is a multiple of 3. This is the convention known as *engineering format*. Compare the left field (scientific) with the right field (engineering) in the following:

```
      a←'¯1.75E5 VS ¯155.75E5'
      a on a ⍒ 2/⍪10*0,⍳8
--------------------
¯1.75E5 VS ¯155.75E5
--------------------
 1.00E0 VS    1.00E0
 1.00E1 VS   10.00E0
 1.00E2 VS  100.00E0
 1.00E3 VS    1.00E3
 1.00E4 VS   10.00E3
 1.00E5 VS  100.00E3
 1.00E6 VS    1.00E6
 1.00E7 VS   10.00E6
 1.00E8 VS  100.00E6
```

8 ***"Check protection."*** Unused positions from here to the decimal point are filled with the character specified in `⎕fc[3]` (which by default is *).

```
      a←'$558,555,535.50'
      a on a⍒4 1⍴123 12345 1234567 123456789
---------------
$558,555,535.50
---------------
  $******123.00
  $***12,345.00
  $1,234,567.00
$123,456,789.00
```

9 ***Conditional zero-fill.*** Like positions marked by `0`, these positions are filled with the character `0` when not otherwise used in the representation of a number. However, when the value being represented is zero, these positions are left blank. In this fashion, an item whose value is zero may be represented by a blank field in the result. (Compare the following with the example for control digit `1`, earlier.)

```
      a←'55,559.9955'
      a on  a ⍒ ⍪ 1 0 1234.1, ÷3
-----------
55,559.9955
-----------
     1.00

 1,234.50
     0.3333
```

Effect of ⎕*fc* on ⍕ Format by Example

The system noun ⎕*fc* is used as an implicit argument to ⍕, affecting the result of its dyadic use with a character left argument. It contains six elements, which are used as follows:

⎕*fc*[1] ***Decimal.*** Character to separate the *integer part* from the *fractional part* of a number. Where the left argument ⍺ contains the character . used as a decimal point, the result of ⍺⍕⍵ substitutes ⎕*fc*[1] for .. In a clear workspace, ⎕*fc*[1] is ..

⎕*fc*[2] ***Grouping.*** Character to separate *groups of digits* in the representation of a number. (The groupings are conventionally but not necessarily triplets.) Where the left argument ⍺ contains , surrounded on *both sides* by digits, the result of ⍺⍕⍵ substitutes ⎕*fc*[2]. In a clear workspace, ⎕*fc*[2] is ,.

⎕*fc*[3] ***Check protection or fill.*** Character to mark *insignificant positions* in the representation of a number. Where the left argument ⍺ contains 8 and the corresponding digit in the result is not significant, the result substitutes ⎕*fc*[3]. In a clear workspace, ⎕*fc*[3] is ⋆.

⎕*fc*[4] ***Field overflow.*** Character to indicate that the representation requires more space than has been specified in the left argument ⍺. In the result, the entire field is replaced by ⎕*fc*[4]. However, when ⎕*fc*[4] is 0 and any of the numbers in ⍵ cannot be represented in the space provided, the entire expression is rejected with a *domain error*. In a clear workspace, ⎕*fc*[4] is ⋆.

⎕*fc*[5] ***Blank.*** Character to indicate where a blank should be substituted in the result. This is useful when you would like the result to have a blank within a numeric field or within a controlled decorator. Where the left argument ⍺ contains this character as a *controlled character* (that is, embedded within the positions devoted to the number's representation), the result contains a blank where the pattern contains ⎕*fc*[5]. In a clear workspace, ⎕*fc*[5] is _.

⎕*fc*[6] ***Negative Sign.*** This position is reserved for a character to indicate a *negative value*. However, at present the only permitted value is ¯. That is the value of ⎕*fc*[6] in a clear workspace.

⍎

Execute

	Monad	*Dyad*	*Identity Element*	*Argument Rank*
⍎	Execute	[undefined]	[none]	⋆ ⋆ ⋆

Only the monad ⍎ is defined. SHARP APL treats the character list (or item) ⍵ as a *line to be executed*, in the same way as a line in the definition of a verb, or a line entered from the keyboard, is executed.

The argument ⍵ may contain only executable characters. In general, executable characters are those enterable from the keyboard. (See "APL Character Set" in Chapter 4, "Nouns and Pronouns"; see also Chapter 15, "APL Characters".)

The result of ⍎⍵ is the result of evaluating the sentence represented by the character list ⍵; for example,

```
      ⍎'2+2'
4

      ⍎'A←3+4'
```

assigns the value 7 to the name A.

The verb ⍎ permits an APL program to construct statements to be executed. Some uses of ⍎ are:

- conditional execution
- conversion of numeric constants
- passing unevaluated arguments to user-defined verbs.

To illustrate *conditional execution*, consider processing a user's response to the question `Proceed to execute Work?` with a statement using ⍎:

```
      ⍎('Yes'≡3↑⍞)/'Work'
```

The verb ⍎ can *convert character lists* representing numeric constants to their numeric values; for example:

```
      a←⍎'1 2 3'
      +/a
6
```

In this rather limited sense, ⍎ is *inverse* to ⍕. (See also the discussion of `⎕fi` and `⎕vi` in Chapter 16, "System Nouns and Verbs".)

Since system commands are *not* APL statements, they *cannot* be executed by ⍎; it follows that the first non-blank character in the argument of ⍎ may *not* be). Nor may ⍎ be used to invoke the ∇-editor, and so the first non-blank character may *not* be ∇.

The verb ⍎ can occur in the sentence that is the argument of ⍎, so that the verb is used recursively.

Result of ⍎⍵

Whether ⍎ has a result depends upon whether the sentence represented in its argument, when evaluated, has a result. When it does, the result of evaluating the sentence becomes the result of ⍎. Otherwise, ⍎ has no explicit result. For example,

```
⍎'¯1+2×⍳⌊0.5×⍴v'
```

returns a result. However, certain primitives have no result (for example, monad ⊣, and system verbs such as □*pdef* or □*tie*). A user-defined verb may be written so that it does not return an explicit result. When the executed sentence has no explicit result, then the verb ⍎ has no result. For example:

```
⍎'⊣¯1+2×⍳⌊0.5×⍴v'
```

[no result]

A sentence that has a result can be embedded in a larger sentence; for example:

```
⍳⍎'¯1+2×⍳⌊0.5×⍴v'
```

A sentence that lacks a result cannot be embedded in a larger sentence:

```
      ⍳⍎'⊣¯1+2×⍳⌊0.5×⍴v'
result error
      ⍳⍎'⊣¯1+2×⍳⌊0.5×⍴v'
        ∧
```

The sentence ⍎'' has no result.

Display of Result

When the verb ⍎ returns a result, the result is displayed if the result would normally be displayed; that is, ⍎'⊢*a*←⍳5' displays a value, but ⍎'*a*←⍳5' does not.

Evaluation of Multiple Sentences

The argument of ⍎ is treated in the same way as a *line* entered from the keyboard or within a user-defined verb. Thus, the argument may contain several sentences delimited by the ⋄ character. It is also permissible to include a final comment to such a line, or an initial label. These are permitted on the principle that the domain of ⍎ should include anything that would be legitimate in the body of a user-defined verb. When the argument of ⍎ contains a label or comment, both the label and the comment are ignored.

Several sentences can be evaluated in a single use of ⍎ if they are included in the argument separated by diamonds; for example:

```
z← ⍎ 'a←b/⍳⍴b ⋄ x←y/⍳⍴y'
```

When the argument of ⍎ contains more than one sentence, the result returned by ⍎ is the value resulting from the *last* sentence evaluated. In the preceding example, `x←y/⍳⍴y` is the last sentence evaluated, and the value assigned to `z` becomes the value of `y/⍳⍴y` in the same way as it would if the sentence were `z←x←y/⍳⍴y`.

Occurrence in the State Indicator

If ⍎ has been invoked but has not completed execution, it appears in the state indicator on a line of its own, in the same way as a user-defined verb. For example, when `fn` is a verb invoked by `⍎'fn'` and its execution is suspended on line 3, then the state indicator appears as:

```
      )si
fn[3] *
⍎
```

In the state indicator, ⍎ is represented by a line containing simply the symbol ⍎. In `⎕lc` (the line counter), there is a corresponding item whose value is the line number of the line that invoked ⍎. Similarly, in the table `2 ⎕ws 2`, there is a row corresponding to a current use of ⍎. (See the discussions of `⎕lc` and also `⎕ws` in Chapter 9, "Control of Execution").

Branch within Execute

When a sentence evaluated by ⍎ contains a branch arrow, → the effect is the same as if the statement had been encountered as part of the defined verb nearest the top of the state indicator.

```
⍎'→ω'  ⟺  →'⍎ω'  ⟺  →⍎'ω'  ⟺  →ω
```

If the value to the right of → is in the domain of →, the branch is taken. (See Chapter 9, "Control of Execution", for further discussion of branching.)

Errors Encountered During Execution

When the sentence you try to execute contains an error, SHARP APL reports an error in the usual way. That includes generating an error message, a display of the offending sentence, and a caret to mark where in the statement it was working when it noticed the error. Execution is suspended on the sentence that contains ⍎, *not* on the line which was the argument to ⍎. However, the sentence displayed in the error message is the argument of ⍎, not the sentence that invoked it.

For example, suppose a user-defined verb called *run* contains on line 4 the sentence *x*←⍎*command*. Suppose that, by the time SHARP APL gets to this statement, *command* is the name of the character list `'fix y'`. But that is not an executable statement because *fix* has not been defined.

In this case, rather than telling you that the error is on line 4 of *run*, SHARP APL shows the statement it is trying to execute (rather than the one that invoked ⍎), thus:

```
value error
run[4]⍎ fix y
        ^
```

The ⍎ symbol is displayed in front of the sentence that ⍎ has been asked to execute. To clarify how the defective sentence was created, the display shows one ⍎ symbol for each level of invocation of ⍎. For example, if the invoking sentence says

```
x←⍎command
```

and the value of *command* was set by

```
command←'⍎''?0'''
```

the error message shows the execute symbol twice, like this:

```
domain error
run[4]⍎⍎ ?0
         ^
```

⎕*signal* within Execute

When ⎕*signal* ⍵ occurs within the argument of ⍎, the effect is to abort execution of ⍎ and signal the event identified in ⍵. The signal is sent to the environment in which `⍎'⎕signal ⍵'` occurred. Notice that the effect of `⍎'⎕signal ⍵'` is *not* the same as the effect of ⎕*signal* ⍵. They differ

both in the activity aborted by `⎕signal` and the environment that receives the signal.

For example, when a user-defined verb `foo` contains the sentence (without ⍎),

```
⎕signal ⍵
```

execution of `foo` is aborted, and event ⍵ is signaled to the sentence that invoked `foo` (and that is therefore next below it on the state indicator).

But when `foo` contains the sentence

```
⍎'⎕signal ⍵'
```

only ⍎ is aborted. The event ⍵ is signaled to the sentence that invoked ⍎ (that is, to the verb `foo`, which appears below ⍎ in the state indicator).

⍪ ,
Table, Ravel; Join

	Monad	***Dyad***	*Identity Element*	*Argument Rank*
⍪	Table	On; Above	[none]	∞ ∞ ∞
,	Ravel	By; Beside	[none]	∞ ∞ ∞

The monad , *ravels* (that is, relaxes the structure) of an existing array. The monad ⍪ forms any array into a table. As dyads, the verbs , (*comma*) and ⍪ (*comma-bar*) produce a single array by joining their two argument arrays.

Monads , and ⍪

The expression ,⍵ ("ravel ⍵") returns a *list* containing all the items of ⍵ in row major order. For example:

```
      t
One
Two
Three

      ,t
One  Two  Three
```

The ravel of an item is a one-element list.

The expression ⍪⍵ ("table ⍵") returns a *table* in which each *row* is the *ravel of a major cell* of ⍵. For example, suppose *X* is a 2-by-3-by-5 array, thus:

```
      X
abcde
fghij
klmno

pqrst
uvwxy
z0123
```

Then ⍪*X* is a 2-by-15 array as follows:

```
     ⍪X
abcdefghijklmno
pqrstuvwxyz0123
```

Similarly, when ⍵ is a 2-by-3-by-4-by-5 array, ⍪⍵ is a 2-by-60 table.

The result of monad ⍪ always has rank 2. When ⍵ is a list, ⍪⍵ is a one-column table. When ⍵ is an item, ⍪⍵ is a 1-by-1 table.

Formally, ⍪⍵ is related to ,⍵ by the identity

```
⍪⍵  ⟺  ,⍤¯1 ,⍤¯1 ⍵
```

in which the second occurrence of ,⍤¯1 is needed only to cover the case when ⍵ is an item.

Figure 6–11 illustrates the effect of ⍪ with the rank conjunction applied in various ways to a rank-5 array. *a* is a 2-by-2-by-2-by-2-by-2 array. Successive sections of the figure show the result and the shape of the result for ⍪⍤5 and for ⍪⍤4, for ⍪⍤3, for ⍪⍤2, and for ⍪⍤1. The result of ⍪⍤0 *a* is not shown. It has shape `2 2 2 2 2 1 1`, and does not fit conveniently on a page, even in small type.

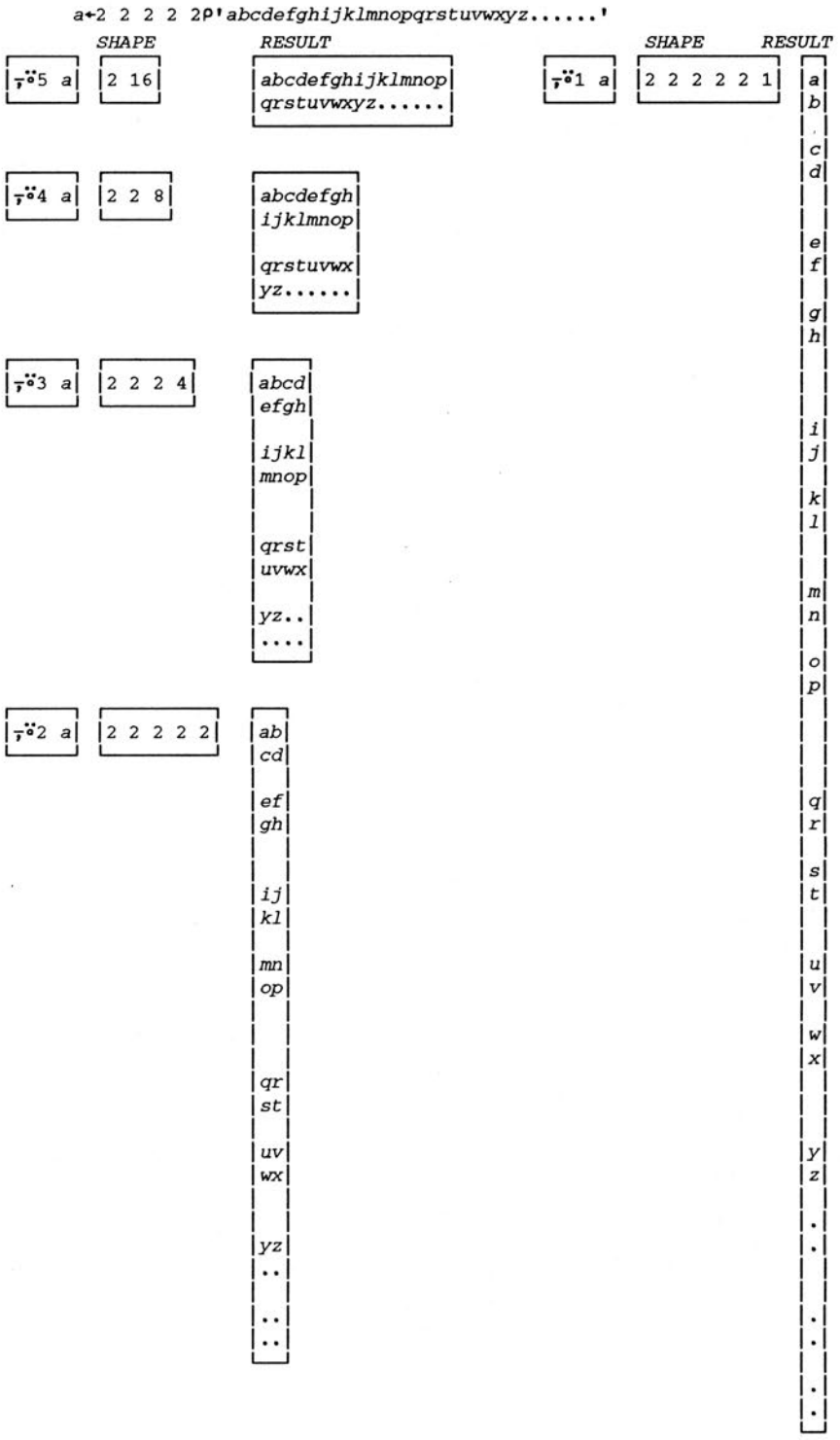


Figure 6–11 : Effect of monad ⍪ *at ranks* 5 *through* 1.

Dyads , and ⍪

The dyad $\alpha⍪\omega$ joins α and ω along their *major* (that is, first) axis. The dyad α,ω also joins two arrays, but along the *last* axis. Applied to tables, $\alpha⍪\omega$ joins them vertically, while α,ω joins them horizontally.

In general, the arrays thus joined must have the same rank. However, their ranks may differ by 1, or one of them may have rank 0.

Dyad , is related to dyad ⍪ by the identities:

$$\alpha,\omega \iff \alpha⍪[\lambda]\omega \iff \lambda(\lambda\alpha)⍪(\lambda\omega)$$

Restrictions on the Domains of ⍪ or ,

The arrays being joined must be of the same type (character, number, or box).[28]

When both arguments are numeric but use different internal representations for numbers (Boolean, integer, floating, or complex), the result has the internal representation of whichever argument has the more extensive representation.

When one of the arrays being joined is empty, the result has the type of the non-empty argument. When both arguments are empty, the result has the type of the right argument.[29]

Shape of Arguments and Result

The arrays being joined must have the same length in all their axes *except* the axis along which they are to be joined. Arrays joined by $\alpha⍪\omega$ must have the same length in all axes except the *first*, and arrays joined by α,ω must have the same length in all axes except the *last*.

If the arrays are not of the same rank, then either one of them must be a scalar, or of rank which differs from the other by one. If they differ by one in rank, the array of lesser rank is treated as if it had length 1 along the axis of joining. If one is a scalar, it is treated as an array of the same shape as the other except along the axis of join, which is treated as length 1. For example:

```
      ⊢ a←2 3⍴'abcdef'
abc
def
```

[28] SHARP APL/UX relaxes this requirement and permits arrays of any type to be joined, producing an array of mixed type.

[29] In principle, the type of an empty array is immaterial. However, the type becomes evident when an empty array is expanded with ↑ or 0\. An empty array may be character or numeric. (An empty array descended from an array of boxes is numeric.)

```
      a,'XY'
abcX
defY
      a,2 1⍴'XY'
abcX
defY
      a⍪'XYZ'
abc
def
XYZ
      a⍪1 3⍴'XYZ'
abc
def
XYZ
      a,'+'
abc+
def+
      a⍪'+'
abc
def
+++
```

The result of $\alpha\overline{,}\omega$ is an array for which the length of the first axis is the sum of the lengths of the first axes of α and ω, and all other axes are unchanged. For example:

```
      b←2 3⍴'uvwxyz'
      b
uvw
xyz

      ⍴a⍪b
4 3

      a⍪b
abc
def
uvw
xyz
```

Similarly, the length of the last axis of α,ω is the sum of the lengths of the last axes of α and ω. All other axes are unchanged.

```
      ⍴a,b
2 6
```

```
      a,b
abcuvw
defxyz
```

Figure 6–12 illustrates ways in which dyad ⍪ may be modified by the rank conjunction. The array *a* is the same as the *a* in the preceding figure, while *A* is an array of the same shape but containing capital letters. The default rank of ⍪ is infinite, so (for these rank-5 arrays) ⍪⍤5 is no different than ⍪. The effect of specifying rank 4, 3, or 2 is illustrated. (In each of the three parts of Figure 6–12, the first box shows the expression executed, the next shows the shape of the result, and the last shows the resulting array.)

Using *on*-rank-4 yields a result whose fourth-from-last axis is the sum of the lengths of that axis in the arguments. Similarly, *on*-rank-3 affects the result's third-from-last axis, and *on*-rank-2 affects the second-from-last. The same principle holds for *on*-rank-1 (see Figure 6–13.), but joining-rank-0 produces a *new* axis in the result.

Joining Two Arrays Along a New Axis

When you join ⍺ and ⍵ along an axis that already exists, you are said to *catenate* them. When you join them along a new axis, you are said to *laminate* them. Catenate and laminate do not refer to different verbs, but to different contexts in which ⍪ or , are used.

To join two arrays along a new axis, they must have the same shape or one must be a scalar. (That is consistent with the general rule that arrays being joined have to have the same length along each axis except the one along which they are joined.)

Assuming that cells occupy the last axis, and all other axes are frame, you can make a new *last* axis by join-rank-0, thus:

```
      'abc' ,⍤0 'xyz'
ax
by
cz
```

In general, when you join two same-shape arrays by ,⍤0 or ⍪⍤0 (it does not matter which), the axes of the result are the (common) axes of ⍺ and ⍵, followed by 2.

There are several ways to join two arrays along a new *first* axis. Both , and ⍪ create a new first axis when their arguments are items. Boxing the arguments causes them to be treated as items. For example:

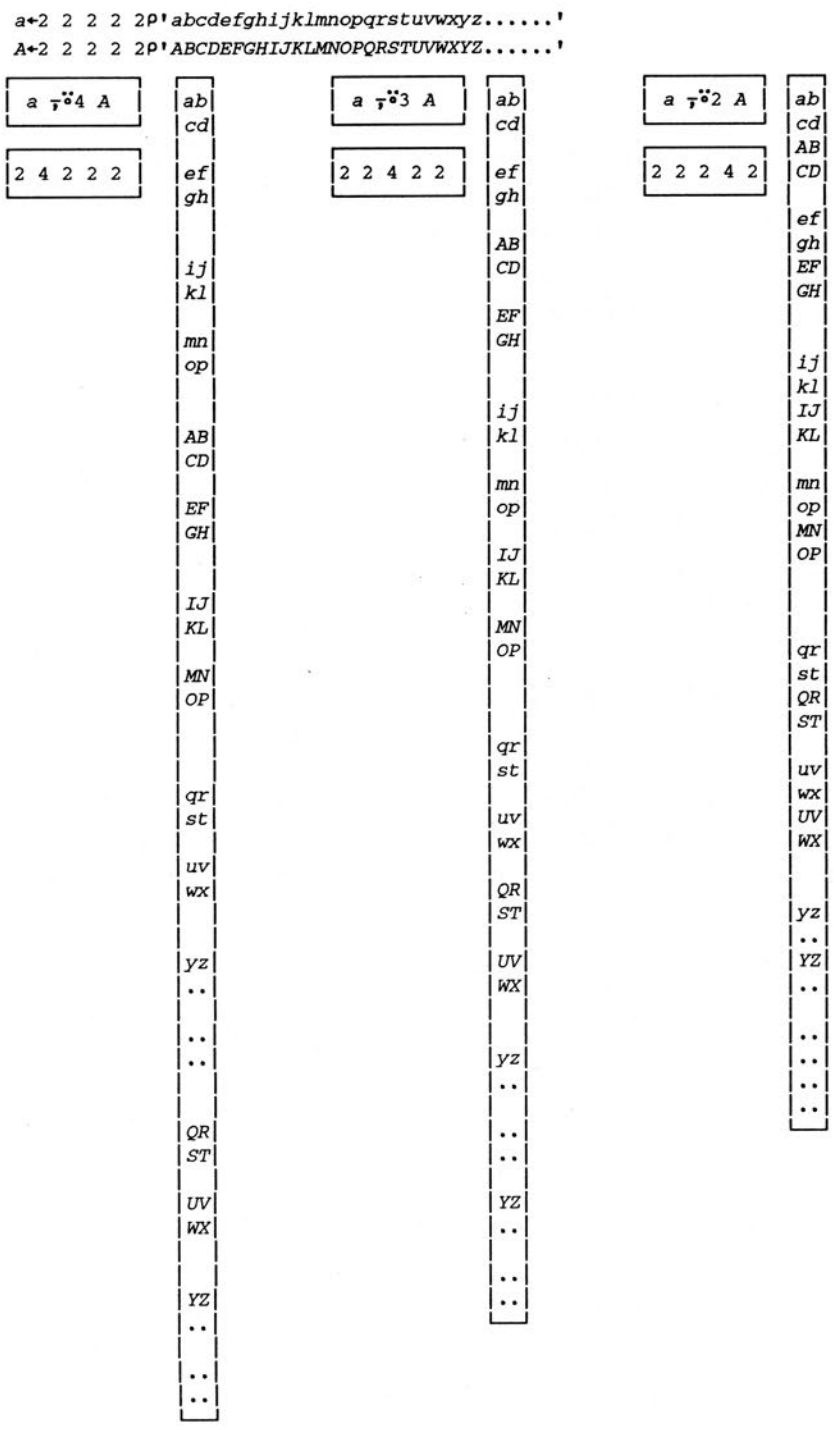


Figure 6–12 : Dyad ⍪ rank 4, 3, and 2.

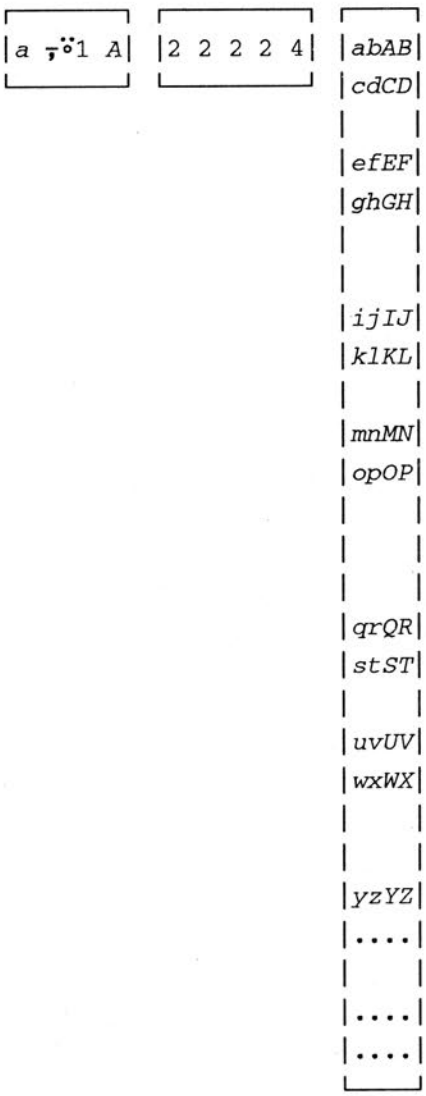


Figure 6–13 : Dyad ⍪ rank 1.

```
      'abc' ,⍤< 'xyz'
abc
xyz
```

The above example applies < monadically to left and right cells. It then applies , to the results thus formed. Finally, the inverse of < (that is, >) is applied monadically to the results formed in the second step.

Or, writing the same thing another way:

```
      > 'abc' ⊃ 'xyz'
abc
xyz
```

Another approach is to increase the rank of one of the arguments before you join them. For example,

```
      >,< ⍵
```

gives ⍵ a new first axis of length 1. (That is because <⍵ is an item, ,<⍵ is the ravel of that item and hence has one axis, and >,<⍵ keeps that new frame-axis when opened.) Then:

```
      'abc'⍪ >,<'xyz'
abc
xyz
```

The monad ⍪ turns a list into a one-column table. Applied to a list, that gives it a new last axis of length 1. If you then transpose it, the new axis is at the front.

```
      'abc' ⍪⍉⍪ 'xyz'
abc
xyz
```

Joined Arrays May Differ in Rank by 1

When one of the arrays has rank 1 less than that of the other, it is treated as if it had an extra axis of length 1. The extra axis is a *leading* axis when the verb is ⍪ and a *trailing* axis when the verb is ,. For example, when you join a table and a list with ⍪, the list becomes a new row in the result; but when you join them with , it becomes a new column. For example:

```
      a⍪'HIJ'
abc
def
HIJ

      a,'12'
abc1
def2
```

Joining an Item to an Array

When one of the arrays is an item (with no axes), the item is replicated to match the lengths of all the axes *other than* the one along which joining occurs:

```
      a⍪'*'
abc
def
***
```

When both arguments are lists, the distinction between ⍪ and , vanishes, since, with only one axis, the first axis *is* the last axis:

```
      'abc'⍪'xyz'
abcxyz

      'abc','xyz'
abcxyz
```

When both arguments are items, the result is a two-item list. (In this case the result has a new axis that was not present in either argument before.)

Joining Modified by Bracket-Axis Notation

Early APL systems permitted certain verbs to be modified by a number in brackets placed after the verb. Because the bracket notation is inconsistent with the rest of APL syntax, this is now regarded as archaic. Nevertheless, to permit existing programs to continue to work without modification, it is still supported in SHARP APL. The verbs , and ⍪ are among those that can be so modified. In the expression

```
α,[k] ω
```

the value inside the brackets is a numeric item that identifies the axis along which the arguments are to be joined. Since it is an index, it is affected by the index origin ⎕*io*. With the axis explicitly identified, α,[*i*]ω has the same meaning as α⍪[*i*]ω.

As usual, the arguments are required to have the same length along the axes other than the axis along which they are to be joined. To join α and ω along axis 2, you write:

```
α,[2]ω
```

To use bracket-axis notation to join along a new axis, you identify the new axis by a fraction that lies somewhere between the existing axis numbers. For example, to join along a new axis to be placed between axis 2 and axis 3, you could write α,[2.5]ω. To join along a new axis to be created ahead of the first axis, α,[0.5]ω.[30] In a 0-origin context, the join could be written as α,[¯0.5]ω).

30 The fraction does not have to be 0.5; any fraction will do provided it differs by less than 1 from an existing axis number.

⊖ ⌽

Reverse, Upset; Rotate, Rowel

	Monad	***Dyad***	*Identity Element*	*Argument Rank*
⌽	Reverse	Rotate	[none]	⋆ ⋆ ⋆
⊖	Upset	Rowel	[none]	⋆ ⋆ ⋆

The monads ⊖ and ⌽ reverse an array's major cells. The dyads ⌽ and ⊖ treat an array as a collection of lists and rotate each list by an amount specified in the left argument.

Monads ⊖ and ⌽

Monad ⊖, called *upset*, reverses the order of the major cells; that is, those at successive positions along the *first* axis. For example, when *t* is a table with three rows, ⊖*t* upsets the order of rows:

```
      t←3 5⍴'One--Two--Three'
      t                                ⊖t
One--                                Three
Two--                                Two--
Three                                One--
```

Monad ⌽, called *reverse*, reverses the order of the 1-cells; that is, those at successive positions along the *last* axis. Monad ⊖ upsets (turns upside down) the order of occurrence of an array's major cells. There is a major cell corresponding to each distinct position along the first axis. In a table, the major cells are the rows of the table. In a rank-3 array, each major cell is a table. In a list, each major cell is an item.

For a rank-3 array *m* of shape 3-by-2-by-16, ⊖*m* upsets the order of the three tables, but leaves the order within each table unchanged:

```
      m                                ⊖m
aaaaaaaaaaaaaaaa                 cccccccccccccccc
AAAAAAAAAAAAAAAA                 CCCCCCCCCCCCCCCC

bbbbbbbbbbbbbbbb                 bbbbbbbbbbbbbbbb
BBBBBBBBBBBBBBBB                 BBBBBBBBBBBBBBBB

cccccccccccccccc                 aaaaaaaaaaaaaaaa
CCCCCCCCCCCCCCCC                 AAAAAAAAAAAAAAAA
```

In similar fashion, monad ϕ reverses the order of the 1-cells. The location of last-axis cells in a table is illustrated in Figure 6–14.

```
      t                ϕ t
O n e - -        - - e n O
T w o - -        - - o w T
T h r e e        e e r h T
```

Figure 6–14 : Reversing the order of a table's last-axis cells.

In a rank-3 array, monad ϕ still reverses cells formed by splitting each position along the last axis. Figure 6–15 shows reversal of the array formed by:

```
t←2 3 5⍴'AppleAvianAxon BasinBeechBroom'
```

```
      t                ϕ t
A p p l e        e l p p A
A v i a n        n a i v A
A x o n          n o x A

B a s i n        n i s a B
B e e c h        h c e e B
B r o o m        m o o r B
```

Figure 6–15 : Reversing the order of last-axis cells in a rank-3 array.

Dyads ϕ and ⊖

The dyads ϕ and ⊖ treat an array as a collection of lists. For each list in ω, α contains a number indicating the amount by which that list is to be rotated. The dyad ϕ, called *rotate*, rotates 1-cells of ω. It behaves as though it has right-argument rank 1. That is, it applies to lists spread along the *last* axis. Dyad ⊖, called *rowel*, does something similar, except that it applies to lists spread along the *first* axis. The symbols are chosen to suggest (at least for a table) the axis about which an array is rotated or reversed.

Each dyad produces a result that has the same shape as ω, with each list at the same location in the overall array. However, *within* each list, the sequence of items is rotated by an amount specified by an item in α which specifies by how much to rotate a list in ω.

α must be an array of integers. In general, it has rank 1 less than the rank of ω. It must have the same shape as ω for all axes *except* the axis along

which rotation takes place, or must be an item. Since dyad ⌽ rotates the last axis, for ⌽, (⍴⍺) must equal ¯1↓⍴⍵. Similarly, since dyad ⊖ rotates the first axis, for ⊖, (⍴⍺) must equal 1↓⍴⍵.

Each list being rotated is treated as a ring. It may be rotated any amount, any number of times. However, rotation by ⍴⍵ simply brings the ring back to where it started, with no change. The effective amount of rotation is modulo the length of the list being rotated. For a seven-item list such as the following, rotation by ¯2 is the same as rotation by 5:

```
      ¯2⌽'abcdefg'
fgabcde

       5⌽'abcdefg'
fgabcde
```

The dyads ⌽ and ⊖ differ only in *where* in ⍵ they find the lists that they rotate. ⌽ rotates the 1-cells of ⍵; that is, lists spread over the *last* axis. ⊖ does the same thing, but to lists spread over the *first* axis. Applied to a rank-1 array, these are the same (since its first axis is also its last axis).

The name *rowel* for ⍺⊖⍵ is chosen by analogy with the toothed gear that by rotating raises or lowers a vertical bar. Applied to a table, dyadic ⊖ requires a left argument list containing an item for *each column* of ⍵. Consider the following 7-by-20 table called `Chart` (shown horizontally double-spaced so that the amount of rotation can be shown for each column):

```
      Chart
. . . . . . . . . . . . . . . . . . . .


* * * * * * * * * * * * * * * * * * * *
⊡ ⊡ ⊡ ⊡ ⊡ ⊡ ⊡ ⊡ ⊡ ⊡ ⊡ ⊡ ⊡ ⊡ ⊡ ⊡ ⊡ ⊡ ⊡ ⊡ ⊡
∘ ∘ ∘ ∘ ∘ ∘ ∘ ∘ ∘ ∘ ∘ ∘ ∘ ∘ ∘ ∘ ∘ ∘ ∘ ∘ ∘ ∘
∘ ∘ ∘ ∘ ∘ ∘ ∘ ∘ ∘ ∘ ∘ ∘ ∘ ∘ ∘ ∘ ∘ ∘ ∘ ∘ ∘ ∘
```

Each column can be rotated independently. For example:

```
0 0 1 1 2 2 3 1 0 ¯2 ¯1 0 0 1 1 2 2 3 3 2⊖Chart
. .         *   .  ∘  ∘ . .         * *
        * * ⊡     ∘  .         * * ⊡ ⊡ *
    * * ⊡ ⊡ ∘ *    .       * * ⊡ ⊡ ∘ ∘ ⊡
* * ⊡ ⊡ ∘ ∘ ∘ ⊡ *       * * ⊡ ⊡ ∘ ∘ ∘ ∘ ∘
⊡ ⊡ ∘ ∘ ∘ ∘ . ∘ ⊡     * ⊡ ⊡ ∘ ∘ ∘ ∘ . . ∘
∘ ∘ ∘ ∘ . .   ∘ ∘  *  ⊡ ∘ ∘ ∘ ∘ . .     .
∘ ∘ . .       . ∘  ⊡  ∘ ∘ ∘ . .
```

In general, the left argument ⍺ must contain an item for each of the first-axis lists of ⍵, so the shape of ⍺ is `1↓⍴⍵`. However, when ⍺ contains a single item, it is applied to all the lists in ⍵:

```
      1⊖t
Two--
Three
One--
```

```
      ¯1⊖t
Three
One--
Two--
```

⌽ and ⊖ Modified by Bracket-Axis

As monads, ⌽ and ⊖ are concerned with reversing cells identified by positions along one axis (the last or the first). As dyads, ⌽ and ⊖ are concerned with rotating each of a set of lists located at positions along one axis (the last or the first). ⌽ takes lists from the last axis, and ⊖ from the first.

The ISO standard for APL permits you to identify some axis other than the first or last, by using brackets. Although bracket-axis notation is now considered obsolescent (because the rank conjunction perhaps combined with transpose now provides a more general and consistent way to achieve the same ends), to permit existing programs to run without modification, SHARP APL continues to support the use of brackets as a conjunction specifying the axis along which reversal or rotation take place, like this:

```
⍺ ⌽[k] ⍵
⍺ ⊖[k] ⍵
```

Since the axis of effect is expressly identified by the value in the brackets, there is no difference between `⌽[k]` and `⊖[k]`. The value inside the brackets must be an integer item and must be a member of `⍳⍴⍴⍵` (and therefore is dependent of the value of the system noun `⎕io`). ⍺ must have rank 1 less than the rank of ⍵ and shape `(k≠⍳⍴⍴⍵)/⍴⍵`.

Upset and Rowel in Terms of Rotate and Reverse

⊖ can be defined in terms of ⌽ by the identities:

$$\ominus\omega \iff \phi^{\bullet\bullet}\otimes\omega$$

$$\alpha\ominus\omega \iff \alpha\phi^{\bullet\bullet}\otimes\omega$$

⍉

Transpose; Cant

	Monad	***Dyad***	*Identity Element*	*Argument Rank*
⍉	Transpose	Cant	[none]	∞ * *

The verb ⍉ *permutes the order of the axes* of an array.

- The left argument of the dyad α⍉ω specifies how the axes are to be permuted. By your choice of left argument, you may also cause certain axes to be mapped together to take a *diagonal slice* through ω.
- The monad ⍉ω reverses the order of the axes of ω.

```
⍉ω  ⟺  (⌽⍳⍴⍴ω)⍉ω
```

When ω is a *list* (having only one axis) or an *item* (having no axes), the result of ⍉ω is ω.

Monad ⍉

When ω is a *table* (rank-2), the result has rows where ω has columns and columns where ω has rows:

```
      ω                ⍉ω
Copenhagen   CLL
Libreville   oiu
Luxembourg   pbx
             ere
             nem
             hvb
             aio
             glu
             elr
             neg
```

The shape of the result of transpose is the reverse of the shape of ω.

```
⍴⍉ω  ⟺  ⌽⍴ω
```

Thus, transposing a rank-3 array moves ω's last axis (columns) so that it becomes the first axis (planes) of the result. When ω is the 2-by-3-by-6 array shown at the left, ⍉ω is the 6-by-3-by-2 array at the right:

```
      ω                    ⍉ω
London                LP
Athens                AP
Ottawa                OM

Prague                or
Peking                te
Moscow                to

                      na
                      hk
                      ts


                      dg
                      ei
                      ac

                      ou
                      nn
                      wo

                      ne
                      sg
                      aw
```

Dyad ⍉

The dyadic form of ⍉ permits you to permute the axes of ω or to map two or more axes together so as to select a diagonal section of ω.

α is a list of integers that tell SHARP APL what to do with each of the axes of ω. α must contain one element for each axis of ω.[31] Transposition has no effect unless ω has at least two axes.

The *positions* within α correspond to the various axes of ω. That is, position 1 within α describes what is to be done with axis 1 of ω, and so on.

The *values* within α refer to the axes of the result. Since axes are numbered starting from `⎕io`, the values in α must reflect the current index origin.

The result can have any rank not greater than the rank of ω. Suppose the desired result rank is called `r`. Then the numbers in α must be chosen such that α includes each member of `⍳r` at least once. For example, when you want a rank-3 result, `1 3 2 1` is a permissible value for α, whereas `1 4 3 4` is not (because it omits 2).

The result returned by α⍉ω is an array formed by permuting or mapping together the axes of ω. The result contains one axis for each unique number in α. When α contains no repetitions, the rank of the result is the same as the rank of ω. Suppose `z←α⍉ω`, then:

31 When ω has rank 1, the system also accepts an *item* for α where, strictly speaking, it should require a one-element list.

```
ρω  ⟷  (ρz)[α]
ρz  ⟷  (ρω)[⍋α]
```

When the items of α are distinct.

Transpositions that Retain All the Axes

When the numbers in α are distinct, all axes of ω are included in the result; only their order is changed. Figure 6–16 illustrates how the axes are rearranged.

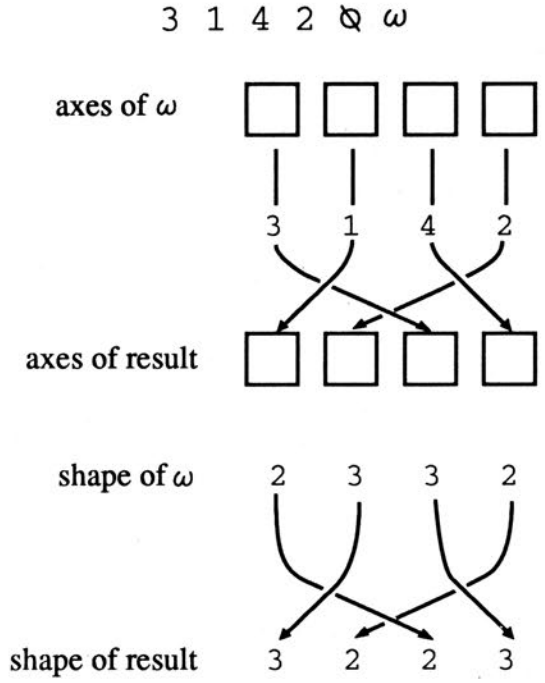


Figure 6–16 : Axes of argument and axes of the result.

Suppose ⍵ has 4 axes. Then the expression

```
3 1 4 2 ⍉⍵
```

says that axis 1 of ⍵ becomes axis 3 of the result;
axis 2 of ⍵ becomes axis 1 of the result;
axis 3 of ⍵ becomes axis 4 of the result;
axis 4 of ⍵ becomes axis 2 of the result.

When ⍺ is a list of consecutive integers in ascending order, the axes are returned in the same positions they had before, and so:

```
(⍳⍴⍴⍵)⍉⍵  ⟺  ⍵
```

When the left argument contains consecutive integers in *reverse order*, the sequence of the axes of ⍵ is reversed, which is the same as the result of monadic transpose:

```
(⌽⍳⍴⍴⍵)⍉⍵  ⟺  ⍉⍵
```

Figure 6–17 illustrates transposition with a specific example, using a rank-4 array whose shape is 2-by-3-by-3-by-2, so that you can trace the relocation of the axes and of each of the items.

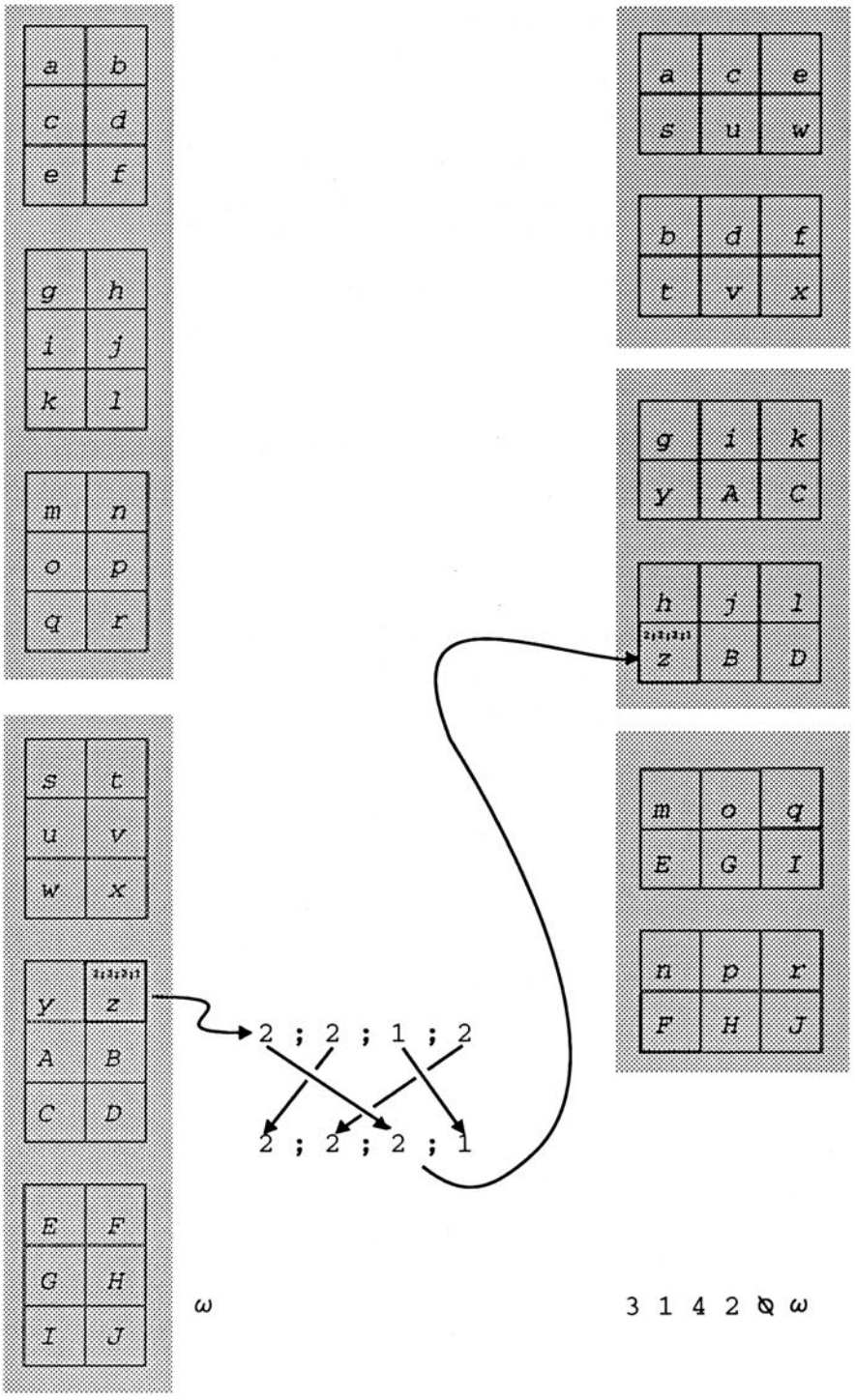


Figure 6–17: Dyadic transpose of a rank-4 array.

Figure 6–18 depicts ω and part of its transpose, showing how the first plane of the first block of the result receives elements from ω. ω is depicted as two blocks of tables, each containing three tables.

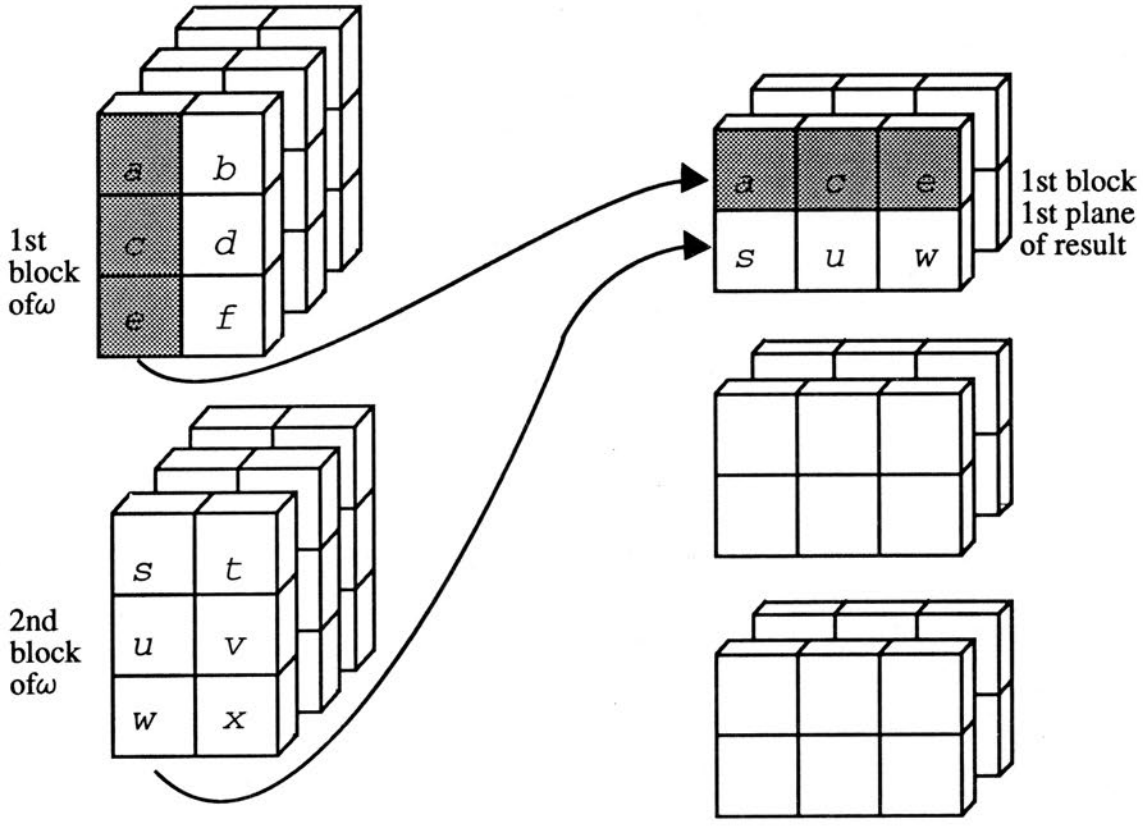


*Figure 6–18 : Visualization of a rank-*4 *array and its* 3 1 4 2 *transpose.*

Diagonal Slices

When α contains repetitions, the number of distinct numbers in α is less than the rank of ω, so the rank of the result is also less than the rank of ω. Consider what happens when ω has rank 4, but α is 1 3 2 3, as in the examples in Figure 6–19 and Figure 6–20. The value 3 occurs both at position 2 *and* at position 4. Axis 3 of the result is to be formed from *both* axis 2 and axis 4 of ω. Two distinct axes of ω are mapped into a single axis of the result.

That is done by taking an item from ω only when its position is the *same* on both axes.

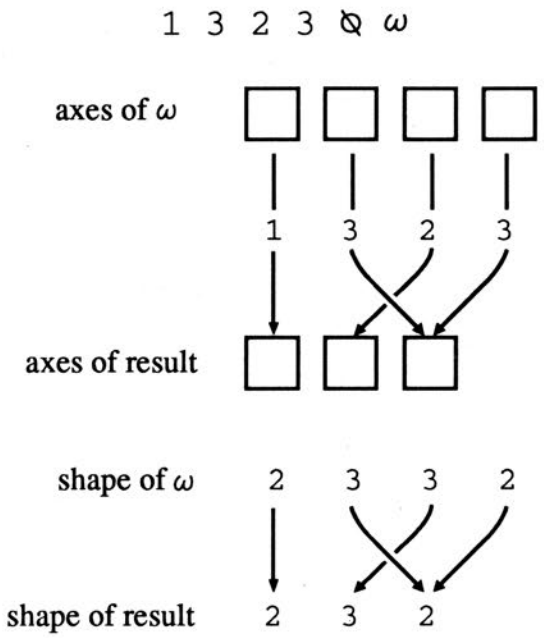


Figure 6–19: Axes mapped together during transpose.

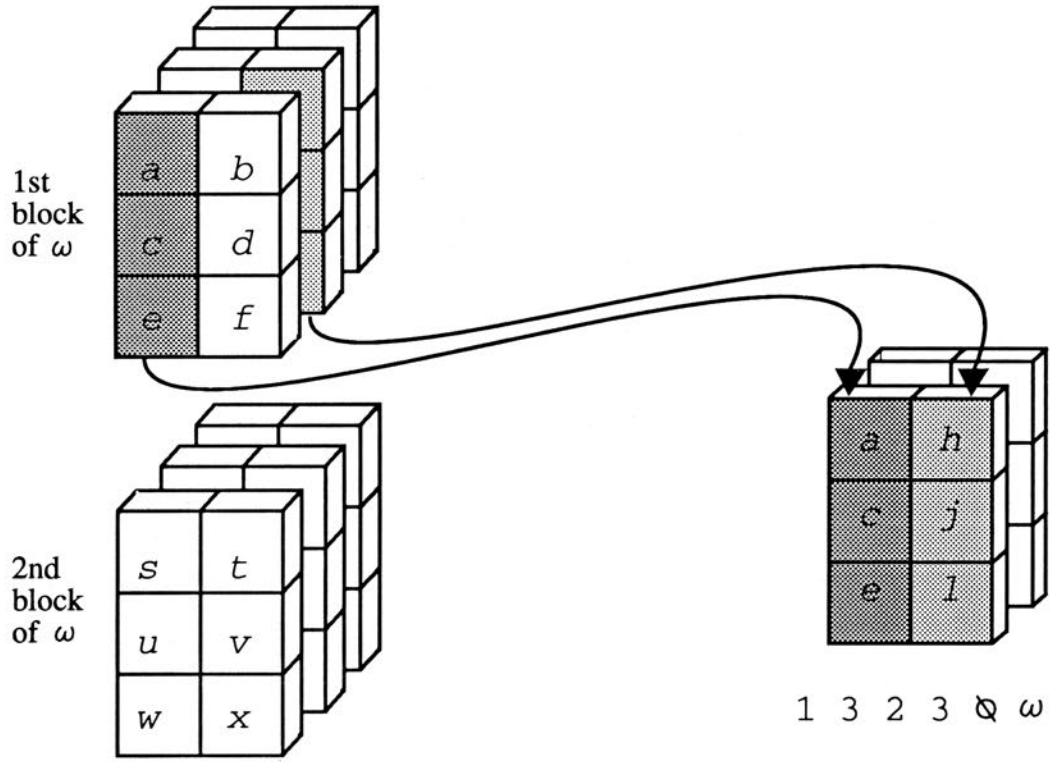


Figure 6–20: Visualization of 1 3 2 3 *transpose.*

By mapping axes together, you are *selecting a diagonal.* For a 4-axis array ⍵, the expression

```
1 1 1 1 ⍉ ⍵
```

gives you the list formed by

⍵[1;1;1;1], ⍵[2;2;2;2], ⍵[3;3;3;3] and so on.

Similarly, for a table ⍵, the transpose `1 1⍉⍵` gives you a list from the main diagonal.

When the axes mapped together differ in length, the number of positions common to both axes is only as great as the length of the *shortest* axis. For example, when ⍵ has shape `3 10`, the expression `1 1⍉⍵` selects only the elements that in Figure 6–21 are shown shaded.

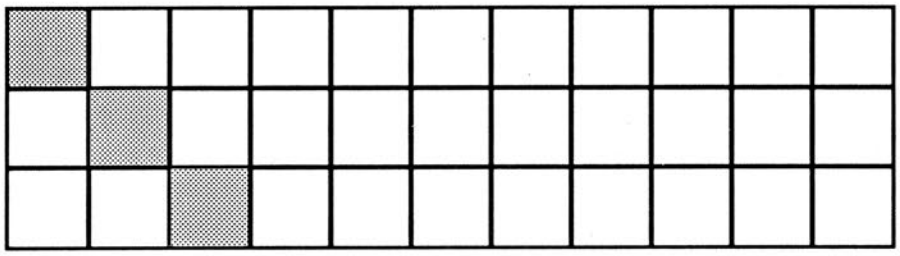


Figure 6–21 : A diagonal is confined to the length of the shorter axis.

Figure 6–22 and the next two figures illustrate several types of diagonal slices taken from a rank-3 array.

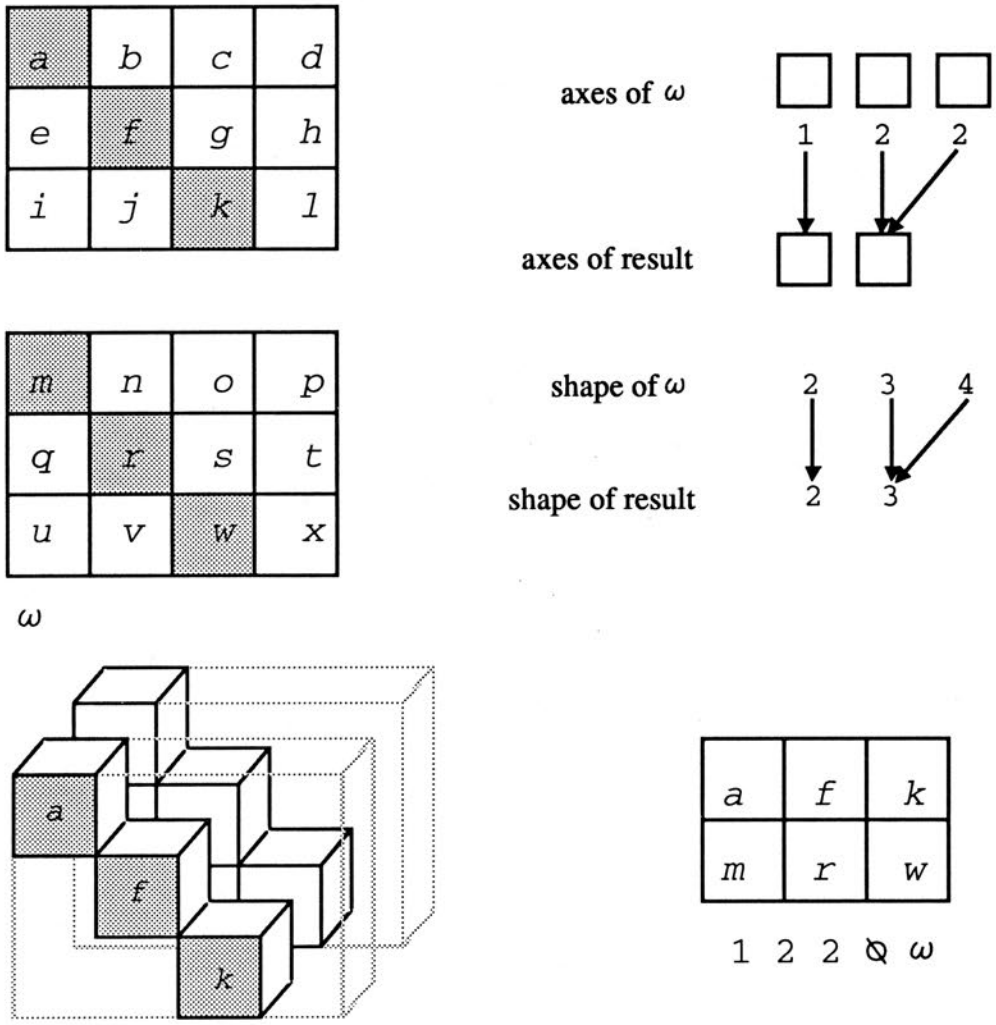


Figure 6–22 : Mapping produced by `1 2 2` *transpose of a rank-3 array.*

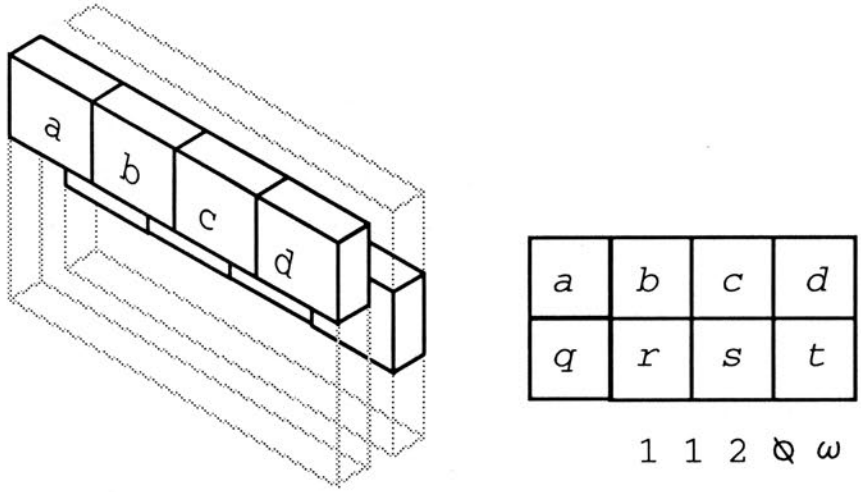
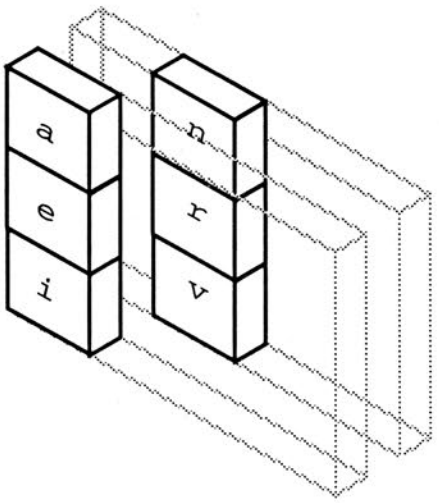


Figure 6–23 : Mapping produced by `1 1 2` *transpose of a rank-3 array.*



a	n
e	r
i	v

2 1 2 ⍉ ⍵

Figure 6–24 : Mapping produced by `2 1 2` *transpose of a rank-3 array.*

↑ ↓

Raze; Take, Drop

	Monad	***Dyad***	*Identity Element*	*Argument Rank*
↑	[none]	Take	[none]	⋆ ⋆ ⋆
↓	Raze	Drop	[none]	∞ ⋆ ⋆

The monad ↓ (*raze*) reconstitutes arrays by assembling the opened elements of the array along the leading axis.

The dyads ↑ (*take*) and ↓ (*drop*) construct an array by selecting a segment along each of the axes of ⍵. The left argument ⍺ is a list that specifies for each axis of ⍵ the length of the segment to be selected.

Monad ↓

Raze opens each item along the leading axis of its argument, catenates those items along the first axis, and boxes them again. Raze is useful for reconstituting data that were separated by the *cut* conjunction. (See Chapter 7, "Adverbs and Conjunctions" for the definition of cut.) Consider the word processing problem of replacing all occurrences of one word by another in a text vector. This might be achieved by cutting the text into words, boxing each word, finding the words of interest, replacing them, and then using raze to paste the words back together into a vector:

```
      ⊢T←2⍤<'This and that and these and those. '
|-----||----||-----||----||------||----||-------|
|This ||and ||that ||and ||these ||and ||those. |
|_____||____||_____||____||______||____||_______|

      T⊣T[(T≡⍤0<'and ')/⍳⍴T]←<'or '
|-----||---||-----||---||------||---||-------|
|This ||or ||that ||or ||these ||or ||those. |
|_____||___||_____||___||______||___||_______|

      >↓T
This or that or these or those.
```

When raze is applied to open arrays, the result is boxed as well:

```
      ⊢m←3 5⍴'fatsoocherbeets'
fatso
ocher
beets
      ↓m
|---||---||---||---||---|
|fob||ace||the||set||ors|
|___||___||___||___||___|
```

When raze is applied to items, the result is the enclosed item.

Operation along other axes can be achieved with the rank conjunction.

Raze of Empty Arrays

Because raze removes elements from the shape vector, it can create non-empty arrays from empty ones. If the leading axis of the argument is of length 0, the result of $\downarrow\omega$ has shape $1\downarrow\rho\omega$. The result is filled with the box fill element.

Raze Along Unit Axis

Raze applied to an argument ω that has length 1 along its leading axis produces a result of shape $1\downarrow\rho\omega$, formed by boxing each item in the argument.

Dyad ↑ and ↓

The dyad *take* describes the result by specifying the segments of ω to be *included* in the result. The dyad *drop* specifies the segments of ω to be *excluded*, so that the result is formed from those that remain.

The result always has the same rank as ⍵ (except when ⍵ is an item). Each value in ⍺ specifies the length of a segment along the *corresponding axis* of ⍵ to be included in or excluded from the result. For example, when ⍵ is a rank-3 array, the expression

```
2 3 4 ↑ ⍵
```

returns

the first 2 positions along ⍵'s first axis;
the first 3 positions along ⍵'s second axis;
the first 4 positions along ⍵'s third axis.

Drop works in a similar manner, except that it omits, or drops, the specified number of items along each axis:

```
2 3 4 ↓ ⍵
```

returns

all but the first 2 positions along ⍵'s first axis;
all but the first 3 positions along ⍵'s second axis;
all but the first 4 positions along ⍵'s third axis.

A *negative* value in ⍺ refers to the *last* positions along an axis. For example, the expression

```
2 ¯3 4 ↑ ⍵
```

returns

the first 2 positions along ⍵'s first axis;
the *last* 3 positions along ⍵'s second axis;
the first 4 positions along ⍵'s third axis.

Again, drop behaves similarly to take with a negative values:

```
2 ¯3 4 ↓ ⍵
```

returns

all but the first 2 positions along ⍵'s first axis;
all but the *last* 3 positions along ⍵'s second axis;
all but the first 4 positions along ⍵'s third axis.

- ⍺ may contain *fewer* elements than the rank of ⍵; in that case, degenerate rules apply, described below.
- ⍺ may never contain more elements than ⍵ has axes, except in the special case when ⍵ has *no* axes, discussed below.
- ⍺ may be an item. In that case it is treated as though it were a one-item list.

See Figure 6–25 for examples of the operation of take and drop.

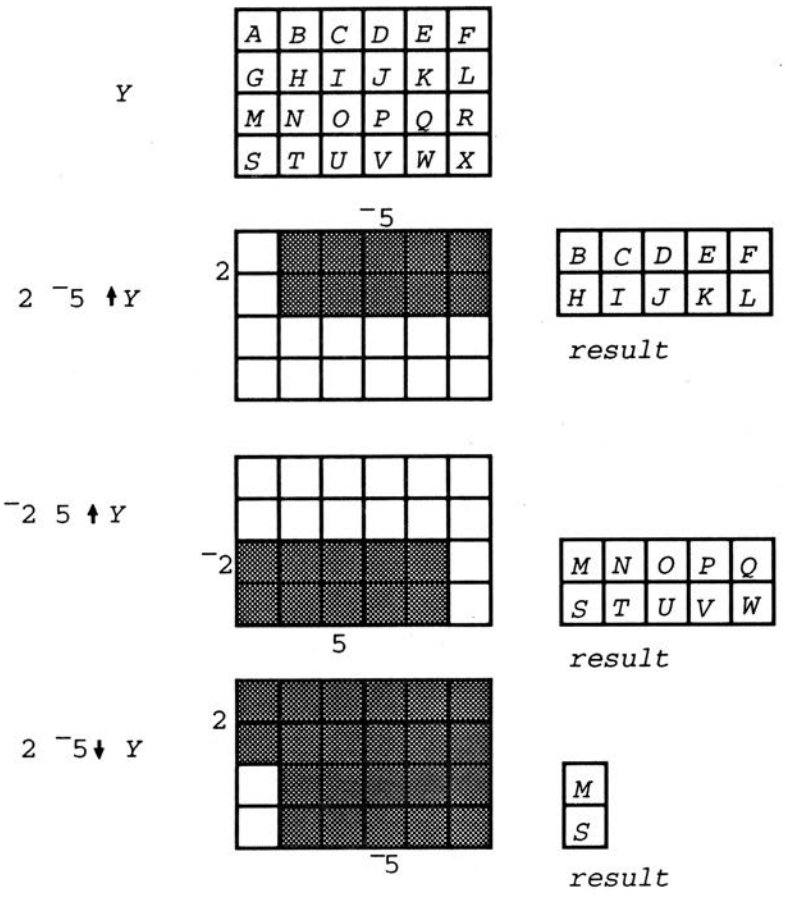


Figure 6–25 : Take and drop.

Take with Short Left Argument

When α has fewer elements than ω has axes, α describes how much to take of the *leading* axes of ω, while the remaining axes are returned entire. Thus, for any array ω have any rank 2 or higher, the phrase

2 ¯3↑ω returns

the first 2 positions along ω's first axis;
the last 3 positions along ω's second axis;
all positions along any remaining axes.

Similarly, the expression 2↑ω returns

the first 2 positions along ω's first axis;
all positions along any remaining axes.

And, by a logical progression, ''↑ω returns all of ω.

In general, the effect of a short left argument is:

α↑ω ⟷ (α,(⍴,α)↓⍴ω)↑ω

Drop with Short Left Argument

When ⍺ has *fewer elements* than ⍵ has axes, ⍺ describes how much to drop from the *leading* axes of ⍵, while the remaining axes are returned without dropping anything. Thus, for an array ⍵ of any rank 2 or greater, the expression

```
2 3↓⍵
```

returns

all but the first 2 positions along ⍵'s first axis;
all but the first 3 positions along ⍵'s second axis;
all positions along any remaining axes.

Similarly, the expression `2↓⍵` returns

all but the first 2 positions along ⍵'s first axis;
all positions along any remaining axes.

And by a logical progression `'' ↓⍵` returns ⍵.

In general, the effect of a short left argument is:

```
⍺↓⍵  ⟺  ((⍴⍴⍵)↑⍺)↓⍵
```

Dropping More Elements than Exist

When ⍺ includes an element whose magnitude is greater than the length of the corresponding axis of ⍵, you are in effect asking to drop more elements than exist along that axis. In the result, the length of that axis is 0; the excess value in ⍺ has no effect. Thus:

```
      ⍴⍵
4 6
      ⍴3 2↓⍵
1 4
      ⍴5 2↓⍵
0 4
```

Over-Take and Padding

When the magnitude of an element of ⍺ is greater than the length of the corresponding axis of ⍵, you are in effect requesting to take from ⍵ more elements than exist along that axis. The result nevertheless contains as many positions as you request. The additional positions contain a *fill* value, which is:

`0`	when ⍵ consists of *numbers*.
`' '`	when ⍵ consists of *characters*.
`<⍳0`	when ⍵ consists of *boxes*.

Where an "oversize" element of α is *positive*, the result contains as many elements as are available in ω, *followed by* the fill value for the remaining positions, so that the fill values (if any) appear at the *end* of that axis.

Where an oversize element of α is *negative*, elements are taken from the *end of that axis* of ω and placed at the end of the corresponding axis in the result. The fill value appears in the extra positions at the *front* of that axis. Suppose ω is the following 3-by-4 table of numbers:

```
1  2  3  4
5  6  7  8
9 10 11 12
```

```
      5 ¯7↑ω
0  0  0  1  2  3  4
0  0  0  5  6  7  8
0  0  0  9 10 11 12
0  0  0  0  0  0  0
0  0  0  0  0  0  0
```

Take When the Right Argument Is Empty

When α calls for creation of a non-empty array but ω is empty, the result contains only fill values. Then the result has the same internal type as the right argument.[32] Although in general the type of an empty array is not significant, information regarding its type is retained and does affect the result produced by take (and also by the derived verb *expand*; see Chapter 7, "Adverbs and Conjunctions").

Type of empty right argument: *character* Result: blanks.

Type of empty right argument: *numeric* Result: zeros.

Take and Drop Applied to Item α

When ω is an item (having no axes), but α is not empty, SHARP APL treats ω as if it had the rank implied by the length of α and every axis had length 1. Thus:

```
      3 ¯7 ↑ 6
0 0 0 0 0 0 6
0 0 0 0 0 0 0
0 0 0 0 0 0 0
```

[32] Note that an empty array is either numeric or character; the value of an empty array descended from an array of boxes is numeric.

```
      ⍴3 ¯7 ↓ 6
0 0
```

Take and Drop Modified by the Rank Conjunction

The rank conjunction specifies the number of axes in an argument cell. Any leading axes left over are frame axes. When the rank conjunction modifies ↑, ⍺ applies not to ⍵ as a whole, but to each cell within ⍵. Thus, the effect of the expression

```
2 3↑⍤1 2 ⍵
```

is to take *for each cell* the first 2 positions along the cell's first axis and the first 3 positions along the cell's second axis. The frame axes are reproduced in the result. Thus, in general:

```
⍺↑⍤(1,k) ⍵  ⟺  ((-k) ↓ ⍴⍵),⍺) ↑ ⍵
```

Similarly, when $k \neq 0$:

```
⍺↓⍤(1,k) ⍵  ⟺  ((-k)↓0×⍴⍵),⍺) ↓ ⍵
```

When *take* is applied with rank 0, each result cell has a new axis for each item in ⍺. In that case, the shape of the result is given by:

```
⍴⍺↑⍤1 0 ⍵  ⟺  (⍴⍵),|⍺
```

And the result is:

```
⍺↑⍤1 0 ⍵  ⟺  ((⍴⍵),|⍺)⍴⍵ ,⍤0 1 (×/|⍺)⍴1↑0⍴⍵
```

⍋ ⍒
Grade up, Grade down

	Monad	***Dyad***	*Identity Element*	*Argument Rank*
	(Numbers)	(Characters)		
⍋	Grade up	Grade up	[none]	∞ ∞ ∞
⍒	Grade down	Grade down	[none]	∞ ∞ ∞

Grade returns a list of integers which if used to select the major cells of ⍵ would arrange them in order. The result has the same length as the first axis of ⍵. The argument must have at least one axis. The result is a *permutation vector*, since it contains each of the first `1↑⍴⍵` integers exactly once. Since the values are indexes, the result is sensitive to `⎕io`.

The verb ⍋ returns the indexes that would arrange the major cells in *ascending* order, while ⍒ returns the indexes that would arrange them in *descending order*. Cells that have equal values are left in the order in which they occur in ⍵. Thus, ⍋ and ⍒ are said to be *stable*.

Monads ⍋ and ⍒

The monads ⍋ and ⍒ are defined for real numbers. Complex numbers have no umambiguous sequence and hence are outside the domain.

The dyads ⍋ and ⍒ are defined only on characters. Characters have no inherent collating sequence, so ⍺ describes the sequence to be used.

The result of any of these verbs is the list of indexes that would sort the major cells of ⍵. When ⍵ is a list, each major cell is an item, and the result of *grade* is the list of indexes that would sort the items. For example:

```
      List←1.2 1.1 1.2 1.4 1.1
      ⍋List
2 5 1 3 4

      List[⍋List]
1.1 1.1 1.2 1.2 1.4

      ⍒List
4 1 3 2 5

      List[⍒List]
1.4 1.2 1.2 1.1 1.1
```

When you grade an array that has more than one axis, the result is still a list of indexes for the *major* cells. The cells are graded by assigning to each cell a numeric value equal to the base value of all the items in the cell, taking them in ravel order (see the definition of , (*ravel*). The base value is evaluated using a radix larger than the value of any of the individual items. That is equivalent to saying that the cells are graded so that the left-most item in the ravel of the cell has greatest weight. For example:

```
      Table
1   4   9   2.1
2   0   0   1.8
1   9   3   0
1   7   7   6

      ⍋Table
1 4 3 2

      Table[⍋Table;]
1   4   9   2.1
1   7   7   6
1   9   3   0
2   0   0   1.8

      Table[⍒Table;]
2   0   0   1.8
1   9   3   0
1   7   7   6
1   4   9   2.1
```

Grade is Not Tolerant

Grade is *not* affected by ⎕ct, comparison tolerance. This is because the monad grade depends upon the transitivity of *greater* or *smaller*, but tolerant comparisons are not transitive.

Since grade is not tolerant, it is possible for grade to reverse the order of two cells even when the cells are reported as equal when compared by the tolerant relation = under non-zero values of ⎕ct.

Dyads ⍋ or ⍒ Grade Character Arrays

To grade character data, you must provide a collating sequence as the left argument of ⍋ or ⍒. SHARP APL has no inherent collating sequence for characters. ⍺ assigns a numeric value to each of the characters in ⍵. With those values, the major cells are graded in the same way as a numeric array.

When the collating sequence ⍺ is a list, the value assigned to each character in ⍵ is ⍺⍳⍵, so:

```
      ⍺⍋⍵ ⍪ ⍋⍺⍳⍵
```

When ⍵ contains a character that appears nowhere in ⍺, it gets a value greater than the value of any of the characters that are present (just as ⍺⍳⍵ returns ⎕*io*+⍴⍺ for an item of ⍵ that is not present in ⍺).

```
      Alf←' abcdefghijklmnopqrstuvwxyz'

      names
jones
baker
jacobs
james
abel

      names[Alf⍋names;]
abel
baker
jacobs
james
jones
```

Collating Sequence with Tied Weights

When the material to be sorted contains both lowercase letters and capitals, what is to be done? Schemes such as *Alf*1 and *Alf*2 include both but do not lead to satisfactory alphabetization. (See Figure 6–26.)

The heart of the matter is that a one axis alphabet such as *Alf* has no way to give two characters the same weight. There is no way to put two characters at the *same* position. The problem is solved by using as many axes as necessary to accommodate sets of characters that are tied.

Suppose you want to place *b* and *B* after *a* or *A* and before *c* or *C*. A simple way to achieve the desired sort is to use a collating sequence with two axes, such as *Alf*3:

```
      '*',Alf3
* abcdefghijklmnopqrstuvwxyz
* ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

(The * is there so you can see that the first position is blank.)

Where ⍺ has more than one axis, the dominant consideration is the position at which a character is found along the *last* axis of ⍺. (In a table, which *column* it is in.) In *Alf*3, both *b* and *B* are at column 3.

```
      Alf1
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
      Alf2
aAbBcCdDeEfFgGhHiIjJkKlLmMnNoOpPqQrRsStTuUvVwWxXyYzZ
      Alf3
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

t	t[Alf1⍋t;]	t[Alf2⍋t;]	t[Alf3⍋t;]
Ama	acid	acid	acid
YMCA	aluminum	aluminum	aluminum
Trudgen	ama	ama	ama
Tektite	ammonia	ammonia	Ama
pi	embolism	Ama	AMA
stroke	pavilion	AMA	ammonia
pavilion	phosphate	DPD	DPD
piping	photosynthesis	embolism	embolism
respiration	pi	NSPF	NSPF
pump	piping	pavilion	pavilion
photosynthesis	plug	phosphate	pH
underwater	pool	photosynthesis	Philodendron
tsunami	porosity	pH	phosphate
pool	pump	pi	photosynthesis
NSPF	pH	piping	pi
recovery	recovery	plug	piping
aluminum	respiration	pool	plug
embolism	stroke	porosity	pool
plug	trudgen	pump	porosity
Tsunami	tsunami	Philodendron	pump
trudgen	underwater	recovery	recovery
pH	Ama	respiration	respiration
porosity	AMA	stroke	stroke
phosphate	DPD	trudgen	Tektite
DPD	NSPF	tsunami	trudgen
ammonia	Philodendron	Tektite	Trudgen
AMA	Tektite	Trudgen	tsunami
Philodendron	Trudgen	Tsunami	Tsunami
acid	Tsunami	underwater	underwater
ama	YMCA	YMCA	YMCA

Figure 6–26 : Effects of alternative left arguments to grade.

A character's location on the other axes contributes to the result only to break a tie between words that otherwise would have the same weight. The next-to-last axis breaks a tie on the last axis; the axis before that breaks a tie on the last two, and so on. To see the effect of this rule, compare the ordering produced by the tabular alphabet `Alf3` with the ordering produced by the single axis alphabet `Alf1`. The list of words to be ordered is taken from the paper by Howard J. Smith, who first proposed the rules for alphabetic grade.[33]

Characters with Identical Weights

The table `Alf3` provides that `b` or `B` comes after `a` or `A` and before `c` or `C`. Wherever there is a tie, a word that contains `b` always precedes a word that contains `B`. However, it is possible to construct the left argument so that certain letters have *identical* weights, and thus words that differ only in those letters are unaffected by grading.

For two characters to have identical weight, they must have the same effective position in α. When the *same* character occurs at two or more positions in α, that character's *effective* position is at the *minimum* of its various coordinates. For example, if the letter `S` occurs at row 1, column 40 and also at row 5, column 13, then its effective position is `1 40⌊5 13`; that is, row 1, column 13. The letter `S` then has identical weight to a unique character located at row 1, column 13, or to to any character that occurs twice, both in row 1 and a higher row, and in column 13 and a higher column.

To give capital and lowercase letters identical weight in sorting, one set (for example, the lowercase letters) is usually placed at the *top left corner* of the alphabet table, while the other set is duplicated, both to the right and below, as illustrated below.

```
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWYXZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

In such a table, the effective weight of `a` is exactly the same as the effective weight of `A`. See Figure 6–27.

[33] H. J. Smith Jr., "Sorting: a New/old Problem", Proceedings of APL79, *APL Quote Quad*, vol. 9, no. 4.

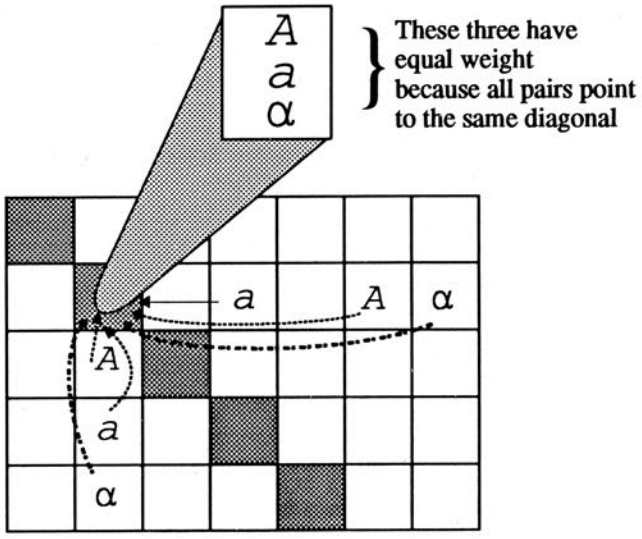


Figure 6–27: Effective weight of characters occurring twice in α.

ρ

Shape; Reshape

	Monad	***Dyad***	*Identity Element*	*Argument Rank*
ρ	Shape	Reshape	[none]	∞ * *

The symbol ρ (*rho*) denotes the verbs *shape* or *reshape*.

- The monad ρω returns the shape of ω.
- The dyad αρω ("α reshape of ω") creates an array whose shape is α and whose items are taken from ω.

Monad ρ

The monad ρω ("shape of ω") returns the length of each of the axes of ω. The result is a list of (non-negative) integers. Since it contains one element for each axis of ω, the number of elements returned tells you the *rank* of ω. The value of each element is the length of the corresponding axis of ω. When ω has no axes, the result is a list containing *no* items (an empty list). For example:

```
      ω←2 3ρ⍳6
      ρω
2 3
      ρρω
2
```

```
      ω←⍳5
      ⍴ω
5
      ⍴⍴ω
1

      ω←6
      ⍴ω
```

[Empty list displays as blank line.]

```
      ⍴⍴ω
0
```

Dyad ⍴

The left argument ⍺ is a list of non-negative integers specifying the length that each axis is to have in the resulting array. SHARP APL accepts an item as equivalent to a one-element list.

The right argument ⍵ is an array of any type or shape. However, ⍵ may be an empty array only when the new array is empty also (that is, when ⍺ contains a zero). SHARP APL rejects an attempt to create a non-empty array from an empty array with the message `length error`.

The expression ⍺⍴⍵ returns an array whose shape is ⍺ and whose items come from ⍵. The positions in the result are filled by taking the elements of ⍵ in *row-major* order. Row-major order describes the sequence in which items in an array are considered. Referring to a table, it means that you take all the items in the first row before you go on to the next row. More generally, for an array of any rank, you step through the positions along the *last* axis, then along the *next to last*, and so on. This is the same order as produced by the APL verb *ravel*, so that:

```
⍺⍴⍵  ⟺  ⍺⍴,⍵
```

In ⍺⍴⍵, when ⍺ asks for *fewer* items than there are in ⍵, items are taken from ⍵ in row-major order until the result is complete; any remaining items of ⍵ are ignored.

When ⍺ asks for *more* items than there are in ⍵, the items of ⍵ (always in row-major order) are repeated cyclically in the result. For example, suppose ⍵ is a 2-by-5 table containing the following values:

```
0 1 2 3 4
5 6 7 8 9
```

Then a result containing *fewer* items than ⍵ might be obtained as follows:

```
      7⍴⍵
0 1 2 3 4 5 6
```

Or by:

```
      2 4⍴⍵
0 1 2 3
4 5 6 7
```

A result requiring *more* items than there are in ⍵ might be obtained as follows:

```
      12⍴⍵
0 1 2 3 4 5 6 7 8 9 0 1
```

Or by:

```
      3 7⍴⍵
0 1 2 3 4 5 6
7 8 9 0 1 2 3
4 5 6 7 8 9 0
```

Note that if ⍺ is a list longer than ⍴⍵ but ending in ⍴⍵, the effect is to replicate ⍵ completely at each of the positions along the new axes. For example:

```
     ⍴⍵
2 5

      3 2 5⍴⍵
0 1 2 3 4
5 6 7 8 9

0 1 2 3 4
5 6 7 8 9

0 1 2 3 4
5 6 7 8 9
```

If the shape of the result differs from the shape of the right argument, a systematic pattern may be generated. For example:

```
      4 4⍴5↑1
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
```

The shape of a single item (a scalar) is empty. To generate an item with dyad ⍴, the left argument ⍺ must be empty:[34]

```
(⍳0)⍴⍵
```

The shape of the result produced by reshape matches the value of ⍺. The only exception is that SHARP APL tolerates a scalar ⍺ for a one-element vector. Thus:

```
⍴⍺⍴⍵  ⟷  ,⍺
```

and

```
(⍴⍵)⍴⍵  ⟷  ⍵
```

∈ Member

	Monad	*Dyad*	*Identity Element*	*Argument Rank*
∈	[none]	Member	[none]	⋆ 0 ∞

For each item (that is, each number, character, or box) within ⍺, the expression ⍺∈⍵ reports whether a matching item can be found anywhere in ⍵. The shape of ⍵ does not matter, nor does the position in ⍵ at which a matching item is found.

The result is a Boolean array having the same shape as ⍺, with a 1 for each item in ⍺ that is matched somewhere in ⍵, and a 0 for each item that is not.

```
      List←2 3 5

      ⊢Table←2 3⍴1 2 3 4 5 6
1 2 3
4 5 6

       2 ∈ List
1

      List ∈ Table
1 1 1

      Table ∈ List
0 1 1
0 1 0
```

[34] For most purposes, the type of an empty array is immaterial, so ''⍴⍵ works equally well.

Items with different levels of boxing never match. For example, if you ask whether the numbers in the open list `1 2 3` are members of an array of boxes, the answer is "no" even when those values are *inside* the boxes:

```
      1 2 3 ∈ (<1 2 3 4),<'abcd'
0 0 0
```

Similarly, a singly enclosed box is not a member of a doubly enclosed box:

```
      ((<'abc'),<'def') ∈ (<<'abc'),<<'def'
0 0
```

Membership Modified by the Rank Conjunction

The rank conjunction ⍤ may organize the frame in which membership in a cell of ω is returned. For example, suppose n is a seven-item list of numbers and T is a nine-row table. (It does not matter how many columns T has.) Then

```
      n
69 3 5 4 8 6 2
      T
7 0 4
6 9 1
1 6 7
1 4 1
5 7 6
0 9 6
1 7 5
8 0 8
3 1 8

      n∈T
0 1 1 1 1 1 0
```

returns a result-frame of rank 1. Within that frame, there are seven cells, each containing a `1` or a `0`, showing whether an element of n exists somewhere in T.

By contrast,

```
      n ∈⍤1 T
0 0 0 1 0 0 0
0 0 0 0 0 1 0
0 0 0 0 0 1 0
0 0 0 1 0 0 0
0 0 1 0 0 1 0
0 0 0 0 0 1 0
```

```
0 0 1 0 0 0 0
0 0 0 0 1 0 0
0 1 0 0 1 0 0
```

says you should consider rank-1 cells from *both* arguments. For `n`, that means the entire array is a single cell, so `n` has an empty frame.

For `T`, that means each row is a cell. There is a frame axis of length 9 (that is, the number of rows that `T` has).

Since one argument has an empty frame and the other has a length-9 frame, the result has a length-9 frame. As usual, the empty frame is paired with *each* of the cells in the other argument. So the first cell of the result is formed from `n∈T[1;]`, the second from `n∈T[2;]`, and so on. Each of those produces a seven-item cell. The cells fit into the length-9 frame, so the result has shape 9-by-7.

∈

Find, In

	Monad	***Dyad***	*Identity Element*	*Argument Rank*
∈	[none]	Find; In	[none]	⋆ ∞ ∞

Dyad ∈

The dyad ∈ (*in*, also known as *string search* or *find*) locates all occurrences of one array within another. For an array ⍵ and a pattern array ⍺, the statement

```
b←⍺∈⍵
```

produces a Boolean array `b` such that the 1s in `b` mark the *beginning points* of the pattern ⍺ in ⍵. The beginning points are the first element of a list, the upper left-hand corner of a table, and so on for higher-dimension arrays. The array `b` has the same shape as ⍵. Both ⍺ and ⍵ can be arrays of any rank.

In can be defined in terms of the *3-cut* of "A Dictionary of APL" as follows:

```
⍺∈⍵ ≡ ((⍳1,-⍴⍺) 3⌺⊂⍵)∈⊂⍺
```

In can be used in string searching as follows:

```
      'to'≤'Onto Toronto, pronto,Tonto!
0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0

      (2 2⍴'toTo')≤⍤1 'Onto Toronto, pronto,Tonto!
0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0
0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
```

In can be used to remove multiple blanks from a string *T* using the expression `(~' '≤T)/T`.

In can be used for pattern finding in higher-rank arrays as well, lending itself to image analysis and similar applications:

```
      ⊢pat←2 3⍴1 2 3 2 3 4
1 2 3
2 3 4
      ⊢image←5 5⍴⍳4
1 2 3 4 1
2 3 4 1 2
3 4 1 2 3
4 1 2 3 4
1 2 3 4 1
      pat≤image
1 0 0 0 0
0 0 0 0 0
0 0 1 0 0
0 1 0 0 0
0 0 0 0 0
```

⍳

Iota; Count, Indexof

	Monad	***Dyad***	*Identity Element*	*Argument Rank*
⍳	Iota; Count	Indexof	[none]	⋆ ⋆ ⋆

The symbol ⍳ is the Greek letter *iota.* Monad ⍳ generates *consecutive integers.* Dyad ⍳ returns *index numbers*: that is, the locations in ⍺ at which the items of ⍵ are found.

Monad ⍳

The monad ⍳ "counts" in the sense that ⍳*n* returns a list of the first *n* integers. The first counting number is either 1 or 0, as set by the system noun ⎕*io*, *index origin*.

```
      ⍳5 ⊣⎕io←1
1 2 3 4 5

      ⍳5 ⊣⎕io←0
0 1 2 3 4
```

Moreover, for a list *L*,

```
      ⍳⍴L
```

is the index for each of the items in *L*, so that

```
      L ⟺ L[⍳⍴L]
```

Default Rank of Monad ⍳

Monad ⍳ expects a single integer as its argument. If you impose argument rank-0, you can generate several sets of consecutive integers.

For consistency with existing applications, SHARP APL treats monad ⍳ as though its usual argument is a list, even though that list may only contain one number. Having a rank-1 argument is handy in expressions such as *A*[⍳⍴*B*].

Forcing argument rank-0 permits a table result, thus:

```
      ⍳⍤0 ⊢ 5 5
1 2 3 4 5
1 2 3 4 5
```

But it also gives a table result in contexts where you may not want it, for example to generate a list of indexes.

Dyad ⍳

Dyad ⍳ (*indexof*) maps the *value* of each item in ⍵ to an *index* that identifies a location in ⍺. The expression ⍺⍳⍵ returns a result that indicates where in the list ⍺ each of the items of ⍵ may be found. For example:

```
      a←'abcdefghijklmnopqrstuvwxyz'
      a⍳'cat'
3 1 20
```

ω may be of any rank or shape. The result has the same rank and shape as ω. For example:

```
      ⍴b
2 3
      b
fat
cat
      a⍳b
 6  1 20
 3  1 20
```

α must be a list, so that a single index is sufficient to identify a location within α.

The result is dependent on ⎕*io*. For example:

```
      a⍳'cat' ⊣ ⎕io←1
3 1 20

      a⍳'cat' ⊣ ⎕io←0
2 0 19
```

For an item in ω that does not occur in α, indexof returns ⎕*io*+⍴α.

For an item in ω that occurs more than once in α, indexof returns the index of its *first* occurrence in ω. This is exploited in a common phrase for computing the *nub* (that is, the unique members) of a list:

```
((⍳⍴List)=List⍳List)/List
```

For an item that is duplicated in *List*, the first occurrence has an index equal to its position. However, later occurrences also have that same index, which therefore must be different from the corresponding member of ⍳⍴*List*, resulting in their exclusion from the nub. Nub can be more tersely defined using *nubsieve*, as (≠ω)/ω.

Tolerant Comparison in Dyad ⍳

The expression α⍳ω asks, in effect, for the index of the first item in α that *matches* an item in ω. *Match* is affected by comparison tolerance, ⎕*ct*. Strictly speaking, indexof returns the first location in α that is *tolerantly equal* to an item in ω. (See the discussion of tolerant comparison in the description of = earlier in this chapter.)

Comparison tolerance can have no effect on the result of indexof or membership when the arguments are characters or integers in the range -2^{31} to $2^{31}-1$. However, it may indeed affect both the results and the time required to compute them for fractional or complex values.

α[ω]
Bracket Indexing

The expression (or expressions) enclosed in square brackets selects items from within the noun α, which may be any array having at least one axis.

Bracket indexing is anomalous in several respects. Unlike other APL verbs, indexing is indicated not by a single symbol but by a pair of symbols. In some respects the left bracket [is the primary symbol, while the right bracket] serves to delimit the right argument. Delimiting the argument implies an *order of execution*, like putting parentheses around the entire indexed expression:

```
a[i] + b  ⟺  (a[i]) + b
```

The brackets surround not an expression but a set of independent expressions, one for each axis of α. To select from an array whose rank is greater than 1 requires more than one index expression; the various expressions are separated from each other by semicolons. When α is a list, one expression suffices to specify positions within it. This is written:

```
α[ω]
```

When α is a list, bracket indexing is inverse to dyadic iota. For an array x (of any rank) all of whose items are contained somewhere in α:

```
x  ⟺  a[a⍳x]
```

However, when α is an array of rank 2 or greater, using bracket indexing to select a sub-array from within it requires a separate expression for each of α's axes. These expressions are separated by semicolons, and the set of expressions for all of α's axes is surrounded by brackets. The number of semicolons is therefore one less than the rank of α. For example:

When α is a table: $\alpha[\omega_1; \omega_2]$

When α has rank 4: $\alpha[\omega_1; \omega_2; \omega_3; \omega_4]$

Figure 6–28 illustrates the use of bracket indexing to select an item from an array.

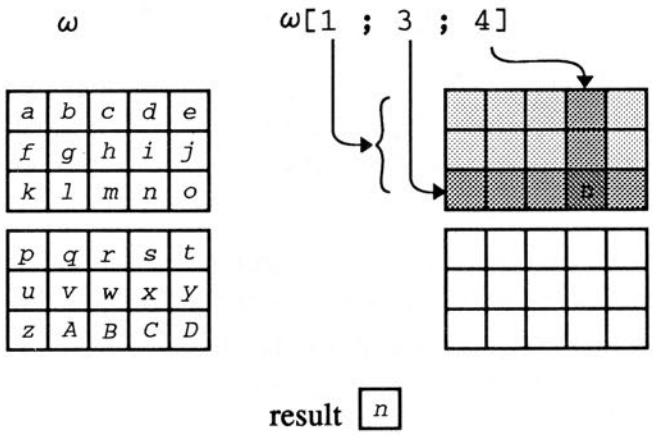


Figure 6–28 : Expression to select an item from a rank-3 array.

Shape of the Result Produced by Bracket Indexing

The result of $\alpha[\omega_1;\omega_2;...]$ is an array formed by selecting items from α at the positions indicated by the values in ω_1, ω_2, etc. Each of the index expressions ω_1 ... ω_n may be of *any* rank or shape, provided only that every one of its elements is a valid position within the corresponding axis of α (subject to the visible value of the index origin $\square io$). The shape of the resulting array is found by catenating the shapes of the indexes for each axis; that is:

$$\rho\alpha[\omega_1;\ \omega_2;\ \ldots\ ;\ \omega_n] \quad \leftrightarrow \quad (\rho\omega_1),\ (\rho\omega_2),\ \ldots\ ,(\rho\omega_n)$$

The symbol ; is punctuation. It is a delimiter, not a verb. Writing $\omega_1;\omega_2$ does *not* form a single array from ω_1 and ω_2. You cannot write a general expression such as $\alpha[\omega]$ that will be valid regardless of the rank of α. In fact, $\alpha[\omega]$ is valid only when α is a list (since it contains zero semicolons), and $\alpha[\omega_1;\omega_2]$ is valid only when α is a table (because it contains one semicolon), and so on. The semicolon acts as an absolute separator in these expressions, so you do *not* need to enclose the various expressions for each axis in parentheses (even when they are compound expressions).

To write a program that can index an array of any rank, you must either generate a character expression containing the correct number of brackets and semicolons and then execute it with ⍎, or else index it by referring always to the ravel of the array, exploiting an identity mentioned below.

Keeping All Positions Along an Axis

You can retain *all* positions along a particular axis of α by *not* selecting along that axis. You do that by writing *no* expression for that axis at the appropriate position between brackets. That axis is then reproduced unchanged in the result.

For example, when α is a list, the result of `α[]` is α.

When α has three axes, then $\alpha[\omega_1;\omega_2;]$ indicates that, along the first axis, positions are to be selected by ω_1; along the second axis by ω_2; but the last axis (for which no index has been specified) is to be reproduced completely in the result. Similarly:

`α[1 2;]` selects positions 1 and 2 (subject to `□io`) from the first axis (rows) and *all* positions from the second axis (columns)

`α[;1 2]` selects *all* positions along the first axis, and positions 1 and 2 (subject to `□io`) along the second axis.

For any axis i for which no ω_i is stated, the shape is `(ρα)[i]`.

The elements selected from α are those whose positions are described by all possible combinations of the first-axis indexes mentioned in ω_1 with the second-axis indexes mentioned in ω_2, and so on for all the axes. See Figure 6–29.

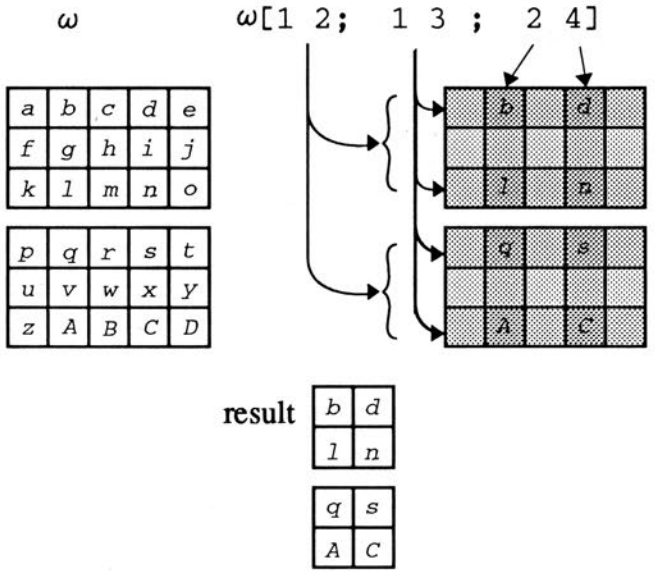


Figure 6–29 : Expression to select a sub-array from a rank-3 array.

Where an axis is indexed by an item (which has *no* axis), the corresponding axis is *omitted* from the result. For example, when ⍺ is a list, ⍺[2] returns an item with no axes because 2 is an item with no axes. See Figure 6–30.

Similarly, when ⍺ is a table, ⍺[1 3 5;2] returns a three-element list, whereas ⍺[1 3 5;,2] returns a 3-by-1 table.

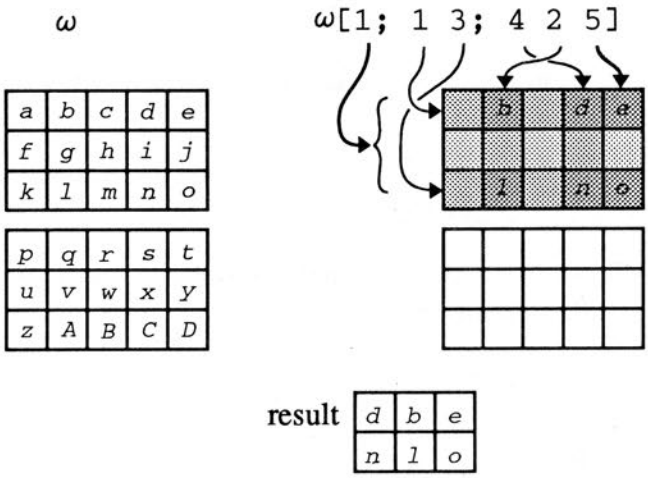


Figure 6–30: Indexing an axis by an item removes that axis from the result.

When any of the selection arrays is present but empty (for example ⍺[ω_1;ω_2] when ω_1 is an empty array), the corresponding axis is present in the result, but has length 0.

Indexed Assignment

A bracket expression ⍺[ω_1;ω_2; ... ;ω_n] may appear to the left of the copula. (See the discussion of indexed assignment in Chapter 5, "Naming Nouns and Pronouns".) Such an expression has the form:

⍺[ω_1; ω_2; ... ; ω_n]←x

Within the brackets, you must have a *valid index expression* or several such expressions separated by semicolons, depending on the rank of ⍺. The requirements for a valid list of indexes are the same as for indexed selection. The expressions inside the brackets imply the shape of a sub-array within ⍺ whose elements are to be *respecified.* The indexes for respecification determine the shape of the sub-array in the same way as the indexes of indexed selection, described in the preceding section. This is also the shape of the result received by any other verb to the left of:

⍺[ω_1;ω_2; ... ;ω_n]←x.

The shape of the right argument usually must match the shape implied by the indexes. However, the shape of the right argument ⍵ is acceptable

if it matches the implied length of each axis having a length other than 1. For example, if the implied shape is `2 3`, then ⍵ could have shape `2 3`, or ⍵ could have shape `1 2 3` or `2 1 3`, or `1 2 1 1 3 1`, and so on. Similarly, when the implied shape is `2 1 3`, the shape of ⍵ would be acceptable if it were `2 3`, or `1 2 3`, and so on. Those axes of ⍵ whose lengths are not 1 must have the lengths implied by the various index expressions within the brackets.

When ⍵ is an item (or one-item array of any rank), SHARP APL reshapes it to match the implied shape.

Effect of Indexed Assignment

SHARP APL replaces the indicated positions within ⍺ with the values of the corresponding elements of ⍵.

```
      ⊢b←2 3 4⍴⍳24
 1  2  3  4
 5  6  7  8
 9 10 11 12

13 14 15 16
17 18 19 20
21 22 23 24

      ⊢b2←2 2⍴101 102 103 104
101 102
103 104

      ⊢b[1 2;3;1 4]
 9 12
21 24

      ⊢b[1 2;3;1 4]←b2
101 102
103 104

      b
  1  2  3   4
  5  6  7   8
101 10 11 102

 13 14 15  16
 17 18 19  20
103 22 23 104
```

The values inserted must be of the same type (character, numeric, or box) as the array into which they are inserted.[35] For numeric arrays, the inserted values may differ from ⍺ in their internal representation (Boolean, integer, floating, or complex). In that case, the entire array ⍺ is converted to the internal type required to accommodate the new elements. This affects the space required to store ⍺. For example, inserting an integer other than `1` or `0` into a Boolean array causes the entire array to be converted to integer internal representation, requiring 32 bits per item rather than one bit per item.

In the case where the indexes contain duplicate values, the right-most value is used. For example:

```
      v[4]⊣v[4 4 4]←1 2 3
3
```

Explicit Result of Indexed Assignment

A sentence may make immediate use of the result of an indexed assignment, perhaps like this:

```
z←⍺[i;]←⍵
```

The result of indexed assignment (like the result of assignment generally) is ⍵, and *not* the merged new value of ⍺. For further discussion, see Chapter 5, "Naming Nouns and Pronouns".

[35] The requirement is relaxed in SHARP APL/UX. There, when ⍺ is initially a homogeneous array, reassigning parts of it with items of different type causes the entire array to be converted to *mixed* type.

7 *Adverbs and Conjunctions*

An adverb modifies the definition of a verb or creates a verb from a noun. A conjunction modifies the definition of a pair of verbs, or a verb and a noun. The modification takes place *before* the verb is executed; what is executed is the modified verb that the adverb or conjunction produces, called the *derived* verb.

Adverb Each of the symbols ⍀ ⌿ ⌽ and ⍂ denotes an *adverb*. An adverb modifies a *single* verb or noun. The adverb symbol follows the symbol for the noun or verb it modifies.

Conjunction Each of the symbols ∘ ⍤ ⍥ and ⍣ denotes a *conjunction*. A conjunction *joins* two verbs to form a new derived verb, or joins a noun and a verb. The verbs that each adverb or conjunction can modify are noted in their descriptions, later in this chapter.

An adverb or conjunction can modify a primitive verb. This chapter opens with a summary of general characteristics of adverbs and conjunctions, and then describes each of the primitive adverbs and conjunctions individually.

Precedence of Adverbs and Conjunctions

APL has one rule of precedence:

- *Adverbs and conjunctions are evaluated before verbs.*

All adverbs and conjunctions have equal precedence (just as, among verbs, all verbs have equal precedence). You have to know what derived verb an adverb or conjunction produces before you can parse the rest of the sentence. Therefore, as you read a sentence from left to right, whenever you come to an adverb or conjunction, you evaluate it at once, and replace it and its arguments with the resulting derived verb.

You treat an adverb or conjunction differently from a verb in the following respects:

- When you come to an adverb or a conjunction, *evaluate it at once.*
- When you come to an unmodified verb, *delay evaluation* until you know the value of its *right argument.* (You do not have to delay for the left argument: because you are reading from left to right, you already know what that is.)

Valence of Adverbs and Conjunctions

An adverb is a *monad.* It applies to the noun or verb to its *left.*

A conjunction is a *dyad.* It applies to the verbs (or the noun and verb) *on each side of it.*

Syntactic Class of the Arguments to an Adverb or Conjunction

The effect of an adverb or conjunction depends upon whether its arguments are nouns or verbs – that is, on the *syntactic class* of its arguments. See the section "Syntactic Classes" in Chapter 3, "Grammar".

- For each *adverb* there are *two* possible meanings, depending upon whether the argument is a noun or a verb.
- For each *conjunction* there are *four* possible meanings, depending upon whether each of the arguments is a noun or a verb.[1]

For most adverbs and conjunctions there is a *generic name* for the symbol (applicable to all the cases) and specific names for the separate meanings that depend on the valence of the arguments.

Ambivalence of the Derived Verb

Once an adverb or conjunction has produced a derived verb, you can use the derived verb either dyadically or monadically. The dyad and the monad are sometimes given distinct names, depending on the effects they produce. The various names are noted as part of the description of each adverb and conjunction.

Consecutive Adverbs and Conjunctions

In principle, a sentence may contain a sequence of several adverbs and conjunctions in succession; "A Dictionary of APL" contains numerous examples. The first adverb modifies a verb to produce a derived verb, which in turn is modified by the next adverb, and so on. Currently, an adverb or conjunction in SHARP APL can apply only to a primitive verb, not to a derived verb, and therefore cannot occur in a sequence such as `+.×/`.

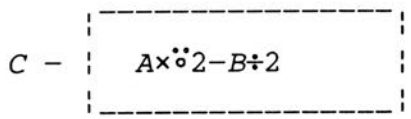
[1] In SHARP APL the noun–noun case does not occur, although it does in "A Dictionary of APL."

Example to Illustrate Order of Evaluation

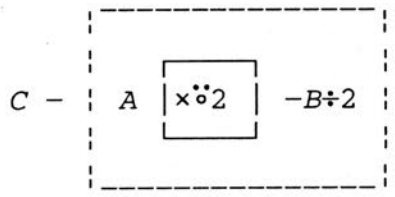
Here is an example that mixes some verbs that are *not* modified by adverbs or conjunctions with some that are. Assume that the names `A`, `B`, and `C` refer to nouns. In the diagrams to the right, a dotted enclosure surrounds an area still to be examined, while a solid box encloses a derived verb.

```
C-A×⍤2-B÷2
```

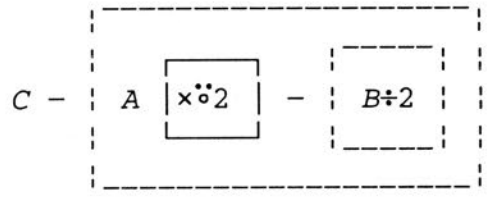
Reading from left to right, the first verb is *subtract*. Since the verb − is not modified by an adverb or a conjunction, it is an ordinary verb. Set it aside until you know what its right argument is.



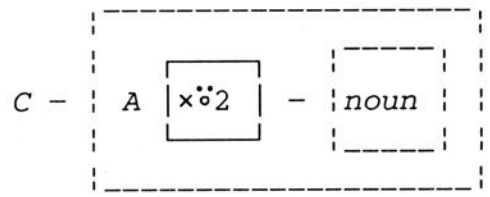
The next verb is ×, which is modified by ⍤, the *rank* conjunction. Evaluate the conjunction at once. Its arguments are the verb × on its left and the number 2 on its right. That gives you the derived verb ×⍤2, read as "times-rank-2." The derived verb ×⍤2 is not modified by another conjunction, but you cannot execute it yet because you do not yet know the value of its argument. Set it aside until its argument is known.



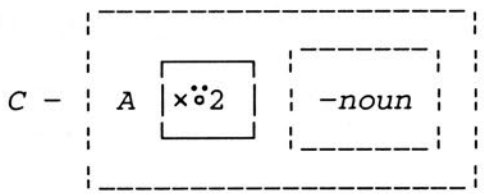
The next verb is −. Since it is not modified by an adverb or conjunction, it is an ordinary verb. Set it aside until you know what its right argument is.



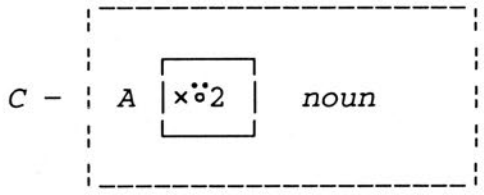
The next verb is ÷. It is not modified. You know the value of its right argument. Evaluate it. Replace `B÷2` with the resulting noun.



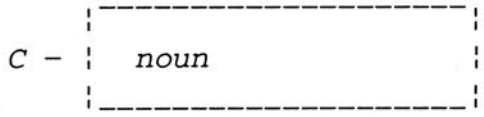
Now you know the value of the right argument of the − that you set aside two steps ago. Go back and evaluate it. It has no left argument, since to its left there is a verb (the derived verb ×⍤2).



Replace the segment *−noun* with the resulting noun.



Now you know the value of the right argument of the derived verb ×⍤2. Evaluate the derived verb. Replace *A*×⍤2−*B*÷2 with the resulting noun.



Now you know the value of the right argument of the − you set aside earlier. Go back and evaluate it. It is the root verb. The sentence has been completely evaluated, and the remaining noun is the result.

Look-Ahead

Reading from left to right requires you to *look ahead.* Each time you come to a noun or verb, you have to check whether it is modified by an adverb or conjunction. You do not have to look far ahead: if there *is* a modifier, it must be the *next* thing to the *right.* Whenever you discover that what is to the right is a noun or verb, that is sufficient to show that your current noun or verb is not modified.

Separating Side-by-Side Constants

Because adverbs and conjunctions have higher precedence than verbs, two nouns may appear side-by-side in a sentence. This happens when both the argument of a conjunction, and the argument of the resulting derived verb are nouns. For example, if you want to find the rank-***n*** product of an array *A* and another array *X*, you write:

```
A ×⍤n X
```

That sentence is perfectly acceptable. But suppose you know that in a particular case *n* is `1` and that *X* is simply the numbers `4 5 6`, and you choose to write those values as constants. The blank, which serves to separate the name *n* from the name *X*, cannot be used to separate the number `1` from the number `4 5 6`. If you were to write

```
A ×⍤1 4 5 6
```

or even

```
A ×⍤1                    4 5 6
```

the numbers would run together to form the single noun

```
1 4 5 6
```

Here are some ways to make plain what you intend:[2]

`A ×⍤1⊢4 5 6`	Use ⊢ to separate the two noun constants. (The verb ⊢ returns the argument to the right without changing it. Its presence makes clear that `1` and `4 5 6` are separate nouns.)
`A ×⍤1(4 5 6)`	Put parentheses around the derived verb's right argument.
`A ×⍤(1)4 5 6`	Put parentheses around the conjunction's right argument.

[2] Although "A Dictionary of APL" allows parentheses around the derived verb, for example `A(×⍤1)4 5 6`, they are not currently permitted in SHARP APL.

Adverbs

v⌿⍵ *v*/⍵ *n*/⍵ *n*⌿⍵
Reduction, Replicate, Compress

v⍀⍵ *v*\⍵ *n*⍀⍵ *n*\⍵
Scan, Expand

These four adverbs produce a family of derived verbs with the following things in common:

Right rank Each has *unlimited right rank*. That is, it applies to the entire right argument at once, rather than independently to multiple cells within a right-argument frame.

Major cells Each partitions its right argument into *major cells*. The adverb whose symbol has a cross bar (that is, ⌿ or ⍀) produces a derived verbs that splits the argument array into major cells along the *first* axis. That is the standard way.

An adverb whose symbol lacks the cross bar (that is, / or \) produces a derived verb that splits the right argument into cells along its *last* axis rather than the first. They are what this manual sometimes calls *contra-major cells*.[3]

The way in which the derived verb uses the cells in its argument is explained in what follows.

Summary of Forms

The argument of the adverb ⌿ or ⍀ may be either a verb or a noun.

Verb argument The verb's *dyadic* use is understood. Although "A Dictionary of APL" permits any verb, in SHARP APL the verb must by a *primitive* and must be a *scalar verb* (that is, a verb whose argument ranks and result rank are all 0). In Figure 7–1, the letter ***v*** stands for the appropriate verb symbol.

Noun argument The noun must be a numeric list. This list is used to control what is done with the first-axis or last-axis cells into which the derived verb partitions its argument. In Figure 7–1, the letter ***n*** stands for the appropriate noun.

[3] Historically, the last-axis split was introduced first, which is why it has the simpler symbols. Recent theoretical work now considers the first-axis definitions to be primary forms and derives the last-axis forms by use of the rank conjunction.

The four forms, with their derived verbs named, and the argument rank of the derived verb that each produces, are summarized in Figure 7–1. (The rank for dyadic use of the derived verb is shown in brackets, since there are no dyadic uses in SHARP APL.)

Form ***v***=Verb ***n***=Noun	Name of the derived verb	Rank of the derived verb
$v \neq \omega$	***v*-reduce** ω	∞ * *
$v \not\backslash \omega$	***v*-scan** ω	∞ * *
$n \neq \omega$	***n*-replicate** ω	∞ * *
	n*-compress** ω (when ***n is Boolean)	
$n \not\backslash \omega$	***n*-expand** ω	∞ * *

Figure 7–1 : Verbs derived from the slash adverbs.

For the corresponding forms with / rather than $\neq$ and with \ rather than $\not\backslash$, use the names shown in the figure qualified by the phrase *last-axis*, as in "last-axis reduce," "last-axis scan," and so on.

Monads Scan and Reduce

When the argument of $\neq$ or $\not\backslash$ is a verb, as in

$$v\neq X \qquad \text{or} \qquad v\not\backslash X$$

the derived verb splits the array X into cells and applies the verb v as a dyad between each of them.

Reduce The result of reduction is found by evaluating the verb ***v*** between *all* the cells thus formed.

Scan The result of a scan is built up as a series of cells by evaluating the verb ***v*** for the first argument cell, the first two argument cells, the first three argument cells, and so on.

Effect of Order of Grouping on Scan and Reduce

Because a scan's n^{th} result cell is constructed from the *first n* cells of the argument, but APL's leaf-to-root execution implies evaluating the *last* verb first, the value of the $(n+1)^{th}$ cell is obtainable directly from the n^{th} cell only when the verb is associative. You can see this in a simple example with the non-associative verb *subtract*.

Example: Suppose X is a numeric array having three axes, with lengths 4 5 6. A scan or reduce splits X along the first axis, forming four cells, each of which is a 5-by-6 table. Call those tables X_1, X_2, X_3, and X_4. Then $-\backslash X$ returns a 4-by-5-by-6 result, whose four tables (each 5-by-6) are computed respectively by:

$$X_1 \quad \Longleftrightarrow \quad X_1$$

$$X_1 - X_2 \quad \Longleftrightarrow \quad X_1 - X_2$$

$$X_1 - X_2 - X_3 \quad \Longleftrightarrow \quad X_1 - (X_2 - X_3)$$

$$X_1 - X_2 - X_3 - X_4 \quad \Longleftrightarrow \quad X_1 - (X_2 - (X_3 - X_4))$$

The left-to-right grouping for the cells produced by a scan was deliberately adopted because, in this order, the scans produced by several non-associative verbs produce valuable results. For instance:

$-\backslash\omega$ *Alternating sum* of cells of ω.

$\div\backslash\omega$ *Alternating product* of cells of ω.

In the result of any scan, the first cell is the same as the first cell in the argument, and the last cell is the same as the result of *reduce*. Thus, the single cell returned by $\neq\backslash\omega$ is the same as the last of the cells returned by $\neq/\omega$.

Where scans and reductions are defined only for scalar verbs (as is at present true in SHARP APL), the result of a scan applied to an array ω has the same rank and shape as the argument ω. The result of a reduction has a rank *one less* than the rank of ω. Indeed, that is why it is called "reduction": it reduces the rank of its argument. (When the argument of reduce already has rank 0, the result also has rank 0.)

Reduce When Argument Has One Cell or None

Since the verb identified in the argument of $\neq$ is interpreted in its *dyadic* sense, you might wonder how the verb is executed when the array argument has only one cell. (What is the sound of one hand clapping?)[4]

By definition, when the array argument of the derived verb has only one cell, the result of reduction by *any* verb is that cell, unchanged. (That holds true even when the values in the argument cell would otherwise be outside the verb's domain, as for example `∧≠'a'`.)

When the array argument of the derived verb has *no cells* and the reduction is by a *primitive* verb with a known *identity element*, then the result is a cell whose value is that identity element. An identity element for a dyadic verb

[4] See also the discussion of identity elements for reduction in the first part of Chapter 6, "Verbs".

is a value that causes the verb to return a result that is the other argument unchanged. Thus, the identity element for addition is 0 because adding 0 to another number does not change it; the identity element for multiplication is 1, and so on.

For ⌈ (maximum), the identity is that value compared to which all other numbers are greater. In principle, it is minus infinity; in practice, it is the smallest representable number. Similarly, for ⌊ (minimum), the identity is that value compared to which all other numbers are smaller. In principle, it is infinity; in practice, it is the largest representable number.

For some verbs, there is a right identity but not a left. For example, anything divided by 1 remains unchanged, but the that is not true for 1 divided by anything. For some verbs, there is an identity element when the verb's domain is restricted. For example, as long as you stick to Boolean values, 1 is an identity element for =.

When reduction is applied to an empty array, SHARP APL returns whatever identity element is defined for the verb. The values of the identity elements are shown in Figure 7–2.

Dyad	Identity Element
+	0
−	0
×	1
÷	1
=	1
≠	0
<	0
>	0
≥	1
≤	1
*	1

Dyad	Identity Element
⊛	None
|	0
!	1
○	None
∧	1
∨	0
⍲	None
⍱	None
⌈	Smallest representable number ¯7.2370055773322621e75
⌊	Largest representable number 7.2370055773322621e75

Figure 7–2 : Identity elements for scalar dyads.

Some Examples of Reduce

Reductions and scans are among the simples, yet most powerful, facilities in APL. The average grade of each student is a course can be found from *S*, their table of quiz scores, by:

```
(+/S)÷¯1↑⍴S
```

The largest and smallest items in each row of an array are found with `⌈/⍵` and `⌊/⍵`, respectively.

Questions relating to Boolean criteria are easily answered with reductions on Booleans:

`∨/bal>1000` ⍝ Does anyone have a balance in excess of $1000?

`∧/parts∈catalog` ⍝ Do we have all the parts required?

`+/graph` ⍝ How many paths exist in the graph?

Some Examples of Scan

Scans, like reductions, are effective tools for a variety of computations:

One way to left-justify a table of text is to count the number of leading blanks in each row, then left-rotate each row of the table by that amount. To do this, use *and scan* to locate the leading blanks, and *plus reduce* to sum them:

```
      ⊢t←4 9⍴'This text is not      yet    aligned'
This text
 is not
    yet
  aligned
      t=' '
0 0 0 0 1 0 0 0 0
1 0 0 1 0 0 0 1 1
1 1 1 1 0 0 0 1 1
1 1 0 0 0 0 0 0 0
      ∧\t=' '
0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0
1 1 1 1 0 0 0 0 0
1 1 0 0 0 0 0 0 0
      +/∧\t=' '
0 1 4 2
      (+/∧\t=' ')⊄t
This text
is not
yet
aligned
```

Sum scan can be used to create a moving average function, such as might be used to filter out high-frequency noise, such day-to-day price fluctuations in a series of stocks. Consider a three-day moving average of *prices*, and the sum scan of *prices*:

```
      prices
8 3 1 8 4 7 8
      ⊢S←+\prices
8 11 12 20 24 31 39
```

If the first two elements of *S* are ignored (three days, remember?), the third item of the sum scan corresponds to the sum for the first three days. The fourth item is the sum of the first four items, the fifth is the sum of the first five, and so on. To get the moving sum, all we have to do is to subtract appropriate numbers from *S*:

0 The sum of the first zero elements of *prices* from the third item.

8 The sum of the first one element of *prices* from the fourth item.

11 The sum of the first two elements of `prices` from the fifth item.

12 The sum of the first three elements of `prices` from the sixth item.

20 And so on.

Noting that these numbers, aside from the first zero, are merely elements of the sum scan, we can complete the moving sum with the following expression:

```
      (2↓S)-0,¯3↓S
12 12 13 19 19
```

The complete expression for the α-period moving average of a list ω is:

```
(((α-1)↓r)-0,(-α)↓r←+\ω)÷α
```

The addition of two rank conjunctions allows the verb to be used on arrays of higher rank:

```
(((α-1)↓⍤1 r)-0,(-α)↓⍤1 r←+\ω)÷α
```

Scans on Boolean arguments are also powerful. ≠\ computes the running parity of a Boolean list. This can be used for such purposes as locating quotes in text. <\ will isolate the first 1 in each row of a Boolean array.

The Derived Verbs Replicate and Expand

When the argument of ≠ or ⍀ is a noun, the derived verb splits its argument into cells and then *copies* (or *replicates*) the argument cells, or expands the argument array by inserting new cells.

n ≠ ω or ***n*** / ω — Makes ***n*** copies of the cells of ω.

- ≠ copies major (first-axis) cells.
- / copies contra-major (last-axis) cells.

n ⍀ ω or ***n*** \ ω — Inserts "fill" cells between existing cells of ω.

- ⍀ inserts major (first-axis) cells.
- \ inserts contra-major (last-axis) cells.

Replicate

The adverb's argument ***n*** indicates the number of times to copy each major cell of the derived verb's argument. Each item in ***n*** must be a non-negative integer, and the length of ***n*** must match the length of the appropriate axis

of ⍵. Alternatively, where ***n*** is an item, it is applied to each of the cells in ⍵.

The number of cells in the result is the sum of the values in ***n*** (or ***n*** times the number of cells, when ***n*** is an item). The result consists of *replications* of the cells of the right argument. For example:

```
      ⊢X← ⍪⍨ 'First' ⊃ 'Second' ⊃ 'Third'
First
Second
Third

      3 0 2⌿X
First
First
First
Third
Third

      2 0 4⌿⍳3 ⊣ ⎕io←1
1 1 3 3 3 3

      3⌿'abc'
aaabbbccc
```

Compress

A commonly used subset of replicate occurs when the values in the list ***n*** are Boolean (that is, are all either `0` or `1`). In such a case, a cell from the derived verb's argument is either retained in the result (where ***n*** has a `1`) or discarded (where ***n*** has a `0`). In that case, the derived verb is called *compress*, since the result differs from the argument only in having certain cells removed. This often occurs in an expression in the form:

(Proposition on ⍵*)* ⌿ ⍵

Here *Proposition on* stands for any APL expression that takes ⍵ as an argument and returns a Boolean result with a `0` or `1` for each major cell in ⍵; for example, `(0∊⍤¯1 ⍵)⌿⍵` ("accept any cell that contains a zero"). The result of this compression retains from ⍵ those major cells for which the proposition is true (represented by `1`) and omits those for which the proposition is false (represented by `0`).

For example, the set of odd numbers could be extracted from a list of numbers using compress in this way:

```
      L
1 9 2 22 55 987 88 100 .5 33333
      (1=2|L)/L
1 9 55 987 33333
```

Expand: Introducing "Fill" Cells

The noun argument to ⍀ or \ is required to be a *Boolean list* in which the number of occurrences of a 1 agrees with the number of cells along the axis of the array you are about to expand. Occurrences of a 0 produce *fill items* appropriate to the *type* of the array you are expanding:[5]

0	for a *numeric* array
' '	for a *character* array
<⍳0	for a *boxed* array.

For example, using the array *X* defined above to illustrate *replicate*:

```
      '*', 1 0 1 0 1 ⍀ X
*First
*
*Second
*
*Third
```

Derived Verbs Modified by Axis Bracket Notation

All the derived verbs produced by ⌿ / ⍀ or \ work by *partitioning* the derived verb's right argument into cells. As mentioned, the derived verbs produced by ⌿ and ⍀ partition along the first axis, while those produced by / and \ partition along the last axis. You can override the axis along which partitioning takes place by modifying the derived verb with the *axis adverb*.[6] For example, the expressions

v⌿[*i*] ⍵	*n*⌿[*i*] ⍵
v/[*i*] ⍵	*n*/[*i*] ⍵
v⍀[*i*] ⍵	*n*⍀[*i*] ⍵
v\[*i*] ⍵	*n*\[*i*] ⍵

5 Fill items are described in detail in Chapter 6, "Verbs" in the sections on "fill elements" and "over-take and padding."

6 Although axis bracket notation does not adhere to the syntax or semantics of adverbs, its behavior is closest to that of an adverb, hence its appearance here. "A Dictionary of APL" treats the use of brackets to indicate an axis as a holdover from early APL implementations and mentions it only as a "dialectical variant." However, it is included in the 1984 ISO standard for APL and is retained in SHARP APL.

all mean that the derived verb partitions the array ω along its i^{th} axis rather than along the first or last. The value of `i` used by the bracket axis adverb is sensitive to the value of `⎕io`.

Conjunctions . ⍤ ⍥ ⍣

For every conjunction, four cases are possible, based on the name-class of the arguments:

- both verbs
- a noun and a verb
- a verb and a noun
- both nouns.

At least in theory, each of these cases may produce an ambivalent derived verb (which may be used either monadically or dyadically). In practice, some of the cases are not implemented or defined. Figure 7–3 summarizes the available cases.

Argument	**Valence**	⍤	⍥	⍣	.
Verb, verb	Monadic	UPON	ON	UNDER	ALTERNANT
	Dyadic	UPON	ON	UNDER	PRODUCT
Noun, verb	Monadic		CUT		
	Dyadic		CUT		
Verb, noun	Monadic		RANK		
	Dyadic		RANK		

Figure 7–3 : Conjunctions.

Grouping in the Discussion of Conjunctions

In the sections that follow, the discussion of conjunctions is grouped not by symbol but by related use. The sections are:

Composition:	⍤ ⍥ ⍣	with arguments ***v*** ***v***
Rank:	⍥	with arguments ***v*** ***n***
Cut:	⍥	with arguments ***n*** ***v***
Product:	.	with arguments ***v*** ***v***

Conjunctions ⍤ ⍥ ⍣ with Verb Arguments

When one of the conjunctions ⍤, ⍥, or ⍣ is used between a pair of verbs, it forms a new verb that is a sort of *composition* of the two verbs. In what follows:

f denotes the verb that is the conjunction's left argument

g denotes the verb that is the conjunction's right argument.

Each of the three conjunctions ⍤, ⍥, and ⍣ may take two verbs as its arguments: one verb which it uses *monadically*, and another which it uses *ambivalently*. Each of the three conjunctions produces an *ambivalent* derived verb. They differ in which of the argument verbs is monadic and which ambivalent, as follows:

f*⍤*g ***Upon.*** The right argument ***g*** is the ambivalent one. It acts on corresponding cells of the derived verb's array arguments ⍺ and ⍵ (or just on cells of ⍵ when ⍺ does not exist).

f*⍥*g ***On.*** The left argument ***f*** is the ambivalent one. It acts on the corresponding cells produced by applying ***g*** to cells of ⍺ and to cells of ⍵ (or just to cells of ⍵ when ⍺ does not exist).

f*⍣*g ***Under.*** Like ***f*⍥*g***. However, to each of the cells produced as in ***f*⍥*g***, it then applies ***g′***, the inverse of ***g***.

Under Requires a Known Inverse

Because ***f*⍣*g*** automatically applies not only the monadic verb ***g*** but also its inverse, the choice of ***g*** in compositions such as ***f*⍣*g*** is limited to verbs for which SHARP APL already knows the inverse. Therefore, ***g*** must be taken from the following list of primitive monads (inverses shown below).

Monad:	⋆	⍟	>	<	−	÷	⍳	○	!	~	⊢	+
Inverse:	⍟	⋆	<	>	−	÷	⍳	○	!	~	⊢	+

Several of these verbs are self-inverse. The verb ⊢ is self-inverse because, used monadically, it is an identity function (giving back as result the same value it received as argument).

Derived Verbs Produced by Composition

In Figure 7–4:

LA denotes the left array argument of the derived verb

α denotes a cell within *LA*

RA denotes the right array argument of the derived verb

ω denotes a cell within *RA*.

Name	APL expression	Valence required	Effect
Upon	*f*⍤*g* *RA*	*f*: 1 *g*: 1	*f g* ω
	LA f⍤*g RA*	*f*: 1 *g*: 2	*f* α *g* ω
On	*f*⍤⍤*g RA*	*f*: 1 *g*: 1	*f g* ω
	LA f⍤⍤*g RA*	*f*: 1 *g*: 2	(*g* α) *f g* ω
Under	*f*⍨*g* *RA*	*f*: 1 *g*: 1	*g*′ *f g* ω
	LA f⍨*g RA*	*f*: 1 *g*: 2	*g*′ (*g* α) *f g* ω

Figure 7–4: Composition templates.

Default Rank of the Derived Verb

A conjunction can accept as a right argument only a primitive verb having a known default argument rank. Thus, a verb whose default argument rank is shown by a ⋆ (in Figure 6–1 or in the descriptions of the various verbs in Chapter 6, "Verbs") is ineligible.

In all cases, the compositions are considered *close*; that is, they are applied *independently to each cell* of their array argument (when they are used monadically) or to each of the *pairs of corresponding cells* (when they are used dyadically).

The verb derived from a composition conjunction divides its argument(s) into cells in the same way as would *g*, the verb to the conjunction's *right*. In particular, the rank of a verb derived from *f*⍤*g* is the same as the rank of *g*, and the rank of verbs derived from either *f*⍨*g* or *f*⍤⍤*g* is the same as the *monadic rank* of *g*.

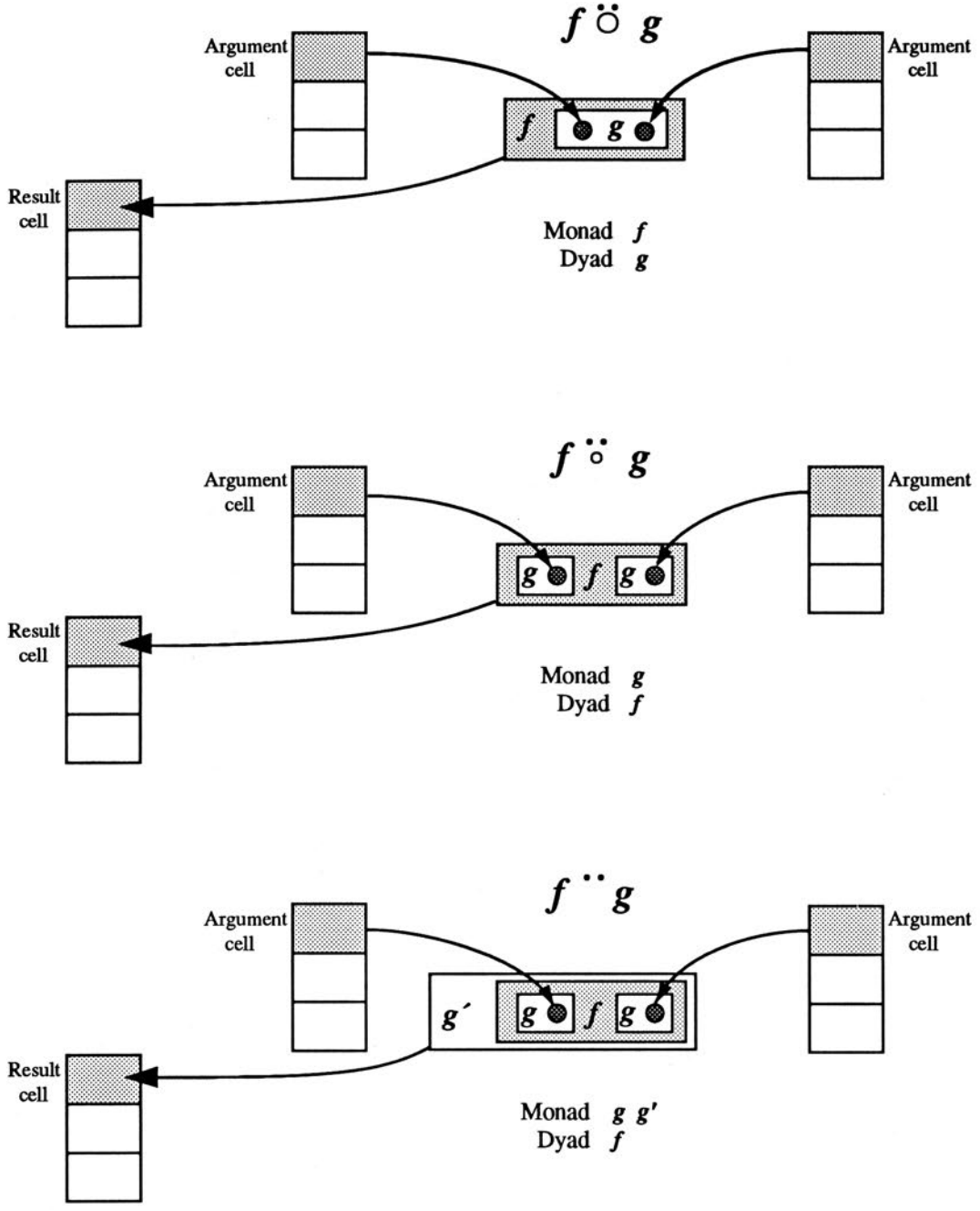


Figure 7–5 : Three forms of composition.

The diagram in Figure 7–5 presents a graphic summary of the way the argument verbs of the three conjunctions are applied to cells of the argument array when the derived verb is used dyadically.

The Rank Conjunction *v*⍤*n*

When the left argument of ⍤ is a verb and the right argument is a noun, the noun specifies the verb's *argument rank*. For example, the expression

```
α +⍤0 2 ω
```

indicates that the verb + is to be applied to rank-0 cells of α and rank-2 cells of ω. That is, each item in α is to be added to the corresponding table in ω. (The frame-shapes have to match. That is, the arrangement of items in α has to match the arrangement of tables in ω.)

Similarly, the expression

```
<⍤1 ω
```

encloses each list (that is, each rank-1 cell) of ω.

The noun argument of ⍤ must be an item or a list, and it may have one, two or three integers. It takes three items to describe the verb's monadic rank and the dyadic ranks of its left and right arguments. When the argument ***n*** has fewer than three items, the values for all three are implied by the expression

```
⌽3⍴⌽n
```

Calling the items within the noun *a*, *b*, and *c* respectively (when the noun has one, two, or three items), they specify the argument ranks as shown in Figure 7–6.

n	Monadic rank	Dyadic left rank	Dyadic right rank
a b c	*a*	*b*	*c*
a b	*b*	*a*	*b*
a	*a*	*a*	*a*

Figure 7–6 : Short forms of rank specification in ⍤.

Argument Rank vs. Complementary Rank

When an item in the right argument of ⍤ is positive, it specifies the number of *trailing* axes in an argument cell. Any remaining axes are frame axes.

When an item in the right argument of ⍤ is negative, its magnitude specifies the number of *leading* axes in the argument frame. Any remaining axes are cell axes. It is thus the complement of the way rank is specified by a non-negative right argument.

For each of the cells within the frame thus defined, the verb is evaluated with its "ordinary rank": as it would be evaluated if not modified by the rank conjunction. For example,

```
<⍤¯1 ⍵
```

specifies that the frame rank is 1, and thus < (*box*) is applied independently to each of the major cells of ⍵.

Within Cells, the "Ordinary" Rank Applies

The rank conjunction controls the initial partitioning of the arguments into frame and cells. To each of those cells, the verb's "ordinary" definition applies – the definition unmodified by the rank conjunction.

When the cells thus defined have a rank greater than the rank for which the verb is defined, then the verb is applied to each cell with its *default rank* – provided default rank is defined for that verb. For example, to add the nine-element list ⍺ to each of the rows of the nine-column table ⍵, you write:

```
⍺ +⍤1 ⍵
```

That pairs the list ⍺ with the list `⍵[1;]`, then with the list `⍵[2;]`, and so on. For the first result cell, the interpreter evaluates:

```
⍺ + ⍵[1;]
```

The verb + had default argument-rank 0. The verb + is then applied to the arguments ⍺ and `⍵[1;]` with rank 0. That causes a new level of partitioning between frame and cell. In this example, each argument cell is partitioned into a nine-element frame consisting of rank-0 cells. `⍺[1]` is added to `(⍵[1;])[1]`, `⍺[2]` is added to `(⍵[1;])[2]`, and so on.

Default Rank Not Defined for Some Verbs

A verb's *default rank* is the argument rank it assumes when the rank is not explicitly stated with the rank conjunction. For example, monadic use

of ⍞ has default rank 2. If x is a rank-3 array and you write ⍞x, it is automatically assumed that you mean to apply ⍞ to each 2-cell of x.

For some verbs, no default rank has yet been defined. For example, monad ⍳ is defined when ω is an item (or a one-item list). If x is a rank-3 array and you write ⍳x, it is *not* assumed that x is to be treated as a collection of rank-0 cells and ⍳ evaluated for each. Instead, the expression is rejected as a *rank error*.

When you use a verb for which default rank has not been defined, the rank conjunction can be used only to assign a rank acceptable in the ordinary case. For example, the expression `⍳⍤0 x` is valid because ⍳ works on a rank-0 argument. And the expression `⍳⍤1 x` works when the last axis of x has length 1 (because ⍳ works on a one-item list, even though it cannot handle a list of any other length).

That is the point of the phrase "ordinary rank" in the preceding section. When you use the rank conjunction to partition the argument into cells, the interpreter applies to each cell the ordinary definition of the verb. When the verb has a default rank, the default rank is used; this may then lead to a further partitioning within each argument cell. But where there is no default rank, the "ordinary rank" – the rank with which the verb is ordinarily used – is the only one permissible. In these cases, you have to supply arguments that meet the verb's requirements for rank and shape.

Examples of Rank

Before the rank conjunction was available, adding a list to each row of a table might have performed with an expression such as:

```
table+(⍴table)⍴list
```

With rank, it is you merely specify the use of rank-1 cells:

```
list+⍤1 table
```

The use of rank allows terser expression as well as offering the potential for faster execution in less space. Adding a list to each column of a table is a bit subtler, but is done as follows:

```
list+⍤0 1 table
```

This takes advantage of scalar extension in +, by setting up scalar cells from the list, and list cells from the table. Effectively, each element of the list is added to the corresponding row of the table.

The Cut Conjunction *n*⍤*v*

When the left argument of ⍤ is a noun (other than 0) and the right argument is a verb, the derived verb *cuts* the argument array ⍵ into segments by splitting it at points along its first axis. A 0-cut selects arbitrary cells from the argument array. Cut then applies the verb on the right to each of the segments thus obtained.

The left argument ***n*** of ⍤ is an integer specifying the *type of cut*. ***n*** *must* be an item; frame considerations do *not* apply to it. The value of ***n*** must be one of ¯2, ¯1, 0, 1 or 2.

When the derived verb is used dyadically, the derived verb's left argument ⍺ specifies the *positions* in ⍵ at which ⍵ is to be cut. To each of the cells produced by a cut, the verb ***v*** is applied *monadically*.

Between them, the derived verb and the conjunction have four arguments; they are summarized in Figure 7–7.

⍺	***n***	⍤	***v***	⍵
Where to cut	*Type of cut*		*Verb* (monad)	*Array* argument of the derived verb
Offset and length *(required)*	0			(to be cut based on the value of the left argument)
Boolean mask *(optional)*	non-0			

Figure 7–7: Arguments of the cut conjunction and of its derived verb.

Types of Cuts

For every type of cut, the derived verb has unlimited right argument rank: it applies to the whole array, not independently to cells within the array. When the derived verb is used dyadically, its left argument rank is either 1 or 2, depending on the type of cut.

The type of cut is controlled by the value of ***n*** as follows:

n=0 Each cell is selected by an *integer table* specifying the *offset* and *length* along each axis.

n≠0 Cells are selected by a *Boolean list* delimiting segments along the *first axis* of ⍵.

Cells Selected by Table

When ***n*** is `0`, the derived verb has left argument rank 2. Each *table* in ⍺ describes the *position and shape* of a cell passed to the verb ***v***. The description is a table having two rows and a column for each axis of ⍵.

Within each ⍺-cell, rows have significance as follows:

Row 1: *Offset* along ⍵'s various axes after which a block is to start. For example, an offset of `2` means that you skip the first two positions before starting the block. A negative offset means "all but." For example, an offset of `¯2` means you skip all but the last two positions before starting a block. An offset of `0` starts at the beginning of an axis; an offset equal to the length of the axis means "start at the end"; it is valid only when the length you ask for is 0. The magnitude of the offset may not exceed the length of the corresponding axis of ⍵.

Row 2: *Length* along each axis for the block you are selecting. The magnitude of the value you write here governs the length of the block. The offset plus the magnitude must indicate positions that actually exist along the corresponding axis of ⍵.

The sign of a value in row 2 affects neither the size of the block you select nor the positions from ⍵ that go into it. The sign controls the *order* in which the selected positions appear in the block. A negative value selects the same positions, but places them in the segment that is passed to the verb ***v*** in reverse order.

For example, here is an ⍺-cell that describes a segment to be cut from a rank-3 array ⍵. Since ⍵ has three axes, an ⍺-cell must have three columns. Thus,

```
4 ¯3  5
2  1 ¯3
```

indicates that from the right argument a cell is to be extracted as follows:

Axis 1: *Offset:* Start four positions after the beginning of this axis.

Length: From there, continue for two positions;

Axis 2: *Offset:* Skip all but the last three positions of this axis.

Length: From there, run for one position in the standard direction (that is, towards the end of the axis).

Axis 3: *Offset:* Start five positions after the beginning of this axis.

Length: From there, run for three positions in the usual direction (that is, towards the end of the axis). However, because the length is negative, place the selected positions in reverse order in the cell passed to verb ***v***.

The 0-cut requires a considerably more elaborate left argument than the 1-cut or 2-cut, with a distinct α-cell for each cell passed to ***v***. However, the 0-cut permits you to specify the lengths a segment has on *all* its axes, not just the first. It also permits *overlapping* segments, or *empty* segments.

The following example shows how multiple pieces may be selected in parallel from a table:

```
      ⊢t←5 5⍴⍳25
 1  2  3  4  5
 6  7  8  9 10
11 12 13 14 15
16 17 18 19 20
21 22 23 24 25

      ⊢c←4 2 2⍴0 0 2 3,0 0 ¯2 ¯3,1 2 3 2,2 ¯2 ¯3 1
 0  0
 2  3

 0  0
¯2 ¯3

 1  2
 3  2

 2 ¯2
¯3  1

      c ⌸<t
|-----| |-----| |-----| |--|
|1 2 3| |8 7 6| | 8  9| |24|
|6 7 8| |3 2 1| |13 14| |19|
|-----| |-----| |18 19| |14|
                |-----| |--|
```

Cells Selected by Boolean Partition

When ***n*** is other than 0, the derived verb has a left argument of rank 1. Each item 1 in the α-cell marks the start (or end) of a cell passed to the verb ***v***. Each Boolean list therefore generates as many cells as it contains 1s. To permit the result to be constructed, each α-cell must contain the same number of 1s. The *box* verb (<) is often used to make this possible.

Each cell has the same rank as ω and the same length as ω along all its axes except the first. Along the first axis, a cell passed to ***v*** has a length

that depends on the distance between consecutive 1s in the α-cell. The positions selected for each cell are controlled by the value of ***n*** as follows:

n∈1 ¯1	Each 1 in an α-cell marks the start of a segment. The first segment starts at the first 1; positions before it are *not* represented in the result.
n∈2 ¯2	Each 1 in an α-cell marks the end of a segment. The last segment ends at the last 1; positions after it are *not* represented in the result.
n positive	The position corresponding to each 1 in an α-cell is included in the result cell.
n negative	The position corresponding to each 1 in an α-cell is not included in the result cell.

To make the selections produced by cut readily visible, use < as the right argument, putting each cell into a box, and make the boundaries of the boxes explicit in the display by setting by setting □*ps* to ¯1 ¯1 ¯3 ¯3. Then α *n*⍤< ω shows how the *n*-cut divides the array argument. For example:

```
      ω←' Worlds on worlds'
      (ω=' ') 1⍤<ω ⊣ □ps←¯1 1 3 3
┌───────┬┬───┬┬───────┐
│ Worlds││ on││ worlds│
└───────┴┴───┴┴───────┘

      (ω∈'Wow') ¯1⍤<ω
┌─┬┬─────┬┬──┬┬─┬┬─────┐
│W│├orlds┤├on┤│w│├orlds┤
└─┴┴─────┴┴──┴┴─┴┴─────┘
```

APL programmers make extensive use of Booleans, particularly in the partitioning of data. One way to easily find the lengths of the partitions in a Boolean partition vector is to let cut do the work for you:

```
      1⍤⍴1 0 0 0 1 1 0 0 1 0
4
1
3
2

      ⊢Table← 5 4⍴⍳20
 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15 16
17 18 19 20
```

```
      (1 0 0 1 0) ¯1⍤< Table ⊣ ⎕ps←1 1 3 3
|----------||----------|
|5  6  7  8||17 18 19 20|
|9 10 11 12||----------|
|----------|
```

Monadic Use of 1– and 2–Cuts

The 1-cut and 2-cut, but not the 0-cut, can be used monadically. When you use the derived verb monadically, the cuts are made by treating the first (or last) of the major cells of ω as the delimiter. The result is what you would get if you had included α as the Boolean list computed as:

```
α←ω ≡⍤¯1 ⊢  >,<⍤¯2 (1↑ω)      when 1=|n
```

or

```
α←ω ≡⍤¯1 ⊢ >,<⍤¯2 (¯1↑ω)      when 2=|n
```

That is, the partition list α is formed by comparing the major cells of ω with the first major cell or the last major cell of ω.

For example, the following encloses each word (delimited by a blank) in the character string ω

```
¯1⍤< ' ',ω
```

while the following arranges each of the equal length character strings between slashes into a table

```
¯1⍤⊢ '/',ω
```

Shape of the Result Produced by Cut

The result produced by dyadic use of a verb derived from cut has the frame-shape required by the values in α, followed (when $n \neq 0$) by the number of partitions, followed by the common cell-shape resulting when the verb v is applied to the cut cells. Because SHARP APL currently requires that result cells agree in rank and shape, the *box* adverb is often used to force all result cells to be items.

Dot Conjunctions

The following table summarizes the cases of the dot conjunction, and the names associated with them. *Outer product*, *Inner product* and *Alternant* are discussed separately in what follows.

Argument	Monadic use of the derived verb	Dyadic use of the derived verb
n ***v***		Outer Product
v ***n***		
v ***v***	Alternant (Determinant)	Inner Product

Figure 7–8 : Verbs derived from the dot conjunction.

Tie and Outer Product

The SHARP APL *outer product* is a particular case of what "A Dictionary of APL" calls *tie*.[7] The outer product relaxes the usual rules regarding agreement of frames, and instead pairs each 0-cell of α in turn with *each* of the 0-cells of ω. The result contains all possible pairings of an α-cell with an ω-cell. The resulting frame has shape `(⍴α),(⍴ω)`.

The outer product is written:

```
α ∘.g ω
```

The argument ***g*** must be a primitive scalar verb (that is, one that is defined for rank-0 arguments and that returns a rank-0 result).

The result is obtained by pairing each element of α with each element of ω. One might specify the result `r` by the following formula:

```
shape ← (⍴α),⍴ω
r ← ((⍴ω)⍴⍤1 0 α) g shape⍴ω
```

Notice that when either α or ω is an item, an outer product gives the same result as the verb without the dot conjunction.

When the arguments are lists, the result is a table. For example, if α is the first three integers and ω the first five, then `α∘.×ω` forms a multiplication table:

[7] In "A Dictionary of APL,"the ***n*** **.** ***v*** case of the dot conjunction relaxes the agreement rules for the frames of a dyad. Whereas ordinarily α and ω must have the *same* frame-shape (and the verb is evaluated for each pairing of an α-cell with the *corresponding* ω-cell), the *tie* conjunction allows you to specify *how many* axes must agree. That leaves all the other (leading) axes free. The ***n*** agreeing axes are said to be *tied*. The result's frame-shape is then the (untied) frame-shape of α followed by the (untied) frame-shape of ω. When the number of tied axes is 0, you have the outer product. As a substitute for `α 0 .g ω`, "A Dictionary of APL" permits `α ∘.g ω`, which is the form provided in SHARP APL.

```
      α ∘.× ω ⊣ α←1 2 3 ⊣ ω←1 2 3 4 5
 1  2  3  4  5
 2  4  6  8 10
 3  6  9 12 15
```

Below appears a division table formed from the same lists α and ω:

```
      α ∘.÷ ω
1          0.5        0.3333333333 0.25       0.2
2          1          0.6666666667 0.5        0.4
3          1.5        1            0.75       0.6
```

The expression α∘.≥ω produces a logical map showing where an element of α is not less than an element of ω:

```
      α ∘.≥ ω
1 0 0 0 0
1 1 0 0 0
1 1 1 0 0
```

Other Boolean verbs also produce interesting results:

```
      α ∘.= ω
1 0 0 0 0
0 1 0 0 0
0 0 1 0 0

      α ∘.< ω
0 1 1 1 1
0 0 1 1 1
0 0 0 1 1
```

Similarly, α ∘.| ω produces a table of residues or remainders:

```
      α ∘.| ω
0 0 0 0 0
1 0 1 0 1
1 2 0 1 2
```

And α ∘.− ω gives a table of the differences between elements of α and those of ω:

```
      α ∘.− ω
 0 ¯1 ¯2 ¯3 ¯4
 1  0 ¯1 ¯2 ¯3
 2  1  0 ¯1 ¯2
```

If `alf` is a list containing the letters of the alphabet, and `text` is a list representing the text of a document, then `alf∘.=text` gives you an incidence map showing where the letters of the alphabet are in the text, while `+/alf∘.=text` gives you the number of occurrences in `text` of each letter in `alf`.

Inner Product

The expression `α+.×ω` is equivalent to the *dot*, *inner*, or *matrix* product as defined in mathematics for vectors (`+/α×ω`) and matrices (where the element in row `i` and column `j` of `α+.×ω` is the dot product of row `i` of `α` and column `j` of `ω`). For example:

```
      1 2 3 +.× 3 4 5
26

      a←2 3⍴⍳6 ⊣ b←3 4⍴⍳12 ⊣ ⎕io←1
      a +.× b
 38  44  50  56
 83  98 113 128

      a +.× 1 2 3
14 32
```

The inner product can be used not just with + and × (the dot product of matrix algebra) but with any of the primitive verbs that take item arguments and return item results. For example:

```
      a ⌊.+ b
2 3 4 5
5 6 7 8

      a +.⌊ b
 6  6  6  6
12 13 14 15
```

Result Shape of Inner Product

The result produced by an inner product resembles the shape produced by outer product, except that the two inner axes – the last axis of `α` and the first axis of `ω` in `α f.g ω` – disappear in the result. Hence, the shape of the result of `α f.g ω` is `(¯1↓⍴α),1↓⍴ω`.

Extending One Argument to Match the Other's Length

In general, the inner axes (that is, the last axis of α and the first axis of ω) must match in length. However, SHARP APL also accepts two other cases:

- When the inner axis of one argument has length 1, but the inner axis of the other argument has some other length, SHARP APL treats the length-1 axis as though it were extended to match the length of the other one, replicating in each position whatever value is in the first position.
- Similarly, when one argument is an item, SHARP APL treats it as if it were a list of the same length as the inner axis of the other argument, replicating at each element the value of the item.

Neither of these extensions affects the shape of the result.

Inner Product and Matrix Product

When α and ω are matrices, the inner product α+.×ω is the same as the ordinary matrix product of matrix algebra. Note that matrix product has an inverse, since, where α and ω are non-singular square matrices:

```
ω  ⟷  (α+.×ω)⌹α
```

See the discussion of matrix division and matrix inverse in the discussion of ⌹ in Chapter 6, "Verbs".

Evaluation of polynomials. The ordinary matrix product α+.×ω can evaluate polynomials in ω. Suppose *coeff* is a table (of any rank or shape) containing the coefficients of the polynomials to be evaluated. The first axis of *coeff* contains the terms, starting with the constant term, the first degree, second degree, and so on, up to the desired degree. Let *powers* be an array in which each member of α is raised to each of the powers, from zero up to whatever degree is implied by the length of the first axis of *coeff*. That can be calculated by:

```
powers←α ∘.* (⍳1↑⍴coeff)-⎕io
```

Then all of the polynomials for each of the sets of coefficients in *coeff* can be calculated by:

```
powers +.× coeff
```

Product of powers. One way to represent an integer is by its prime factorization. That is, you can represent an integer by a list indicating to what power each of its possible prime factors must be raised. The integer is then the product over each of those powers. For all the primes in order out to the largest one needed, suppose *factors* is an array containing the power

to which each must be raised. Then the values represented by such a set of prime factors are found by

```
primes ×.* factors
```

where *primes* is a list of the values of as many primes as are needed to match the length of the first axis of *factors*.

Locating characters from one table within another. Inner products constructed from logical verbs on character data are frequently useful in handling textual data. Suppose *sample* is a character matrix in which each row contains characters spelling a word, phrase, or name. Suppose *reference* is a similar table of recognized words or phrases. You need to know where within *reference* each row of *sample* is to be found. A map showing where each row of *sample* is entirely equal to a row of *reference* is obtained by:

```
reference ∧.= ⍉sample
```

Alternatively, rather than identifying those rows of *reference* which are entirely equal, you might wish to eliminate those that are in any way different, by an expression such as:

```
reference ∨.≠ ⍉sample
```

If you want to convert the resulting Boolean map so as to identify the number of the row on which each match was found, that might involve another inner product. Suppose *iota* is a list of consecutive integers, with as many members as there are rows in *reference*, perhaps formed by the expression:

```
iota←⍳1↑⍴reference
```

The location of the match for each row of *sample* is found by:

```
iota +.× reference ∧.= ⍉sample
```

The preceding formula assumes that *reference* does not contain duplicate entries. But if it did, and you wanted the highest entry, you could find it as:

```
iota ⌈.× reference∧.=⍉sample
```

If you are interested in the extent of agreement (rather than simply the Boolean decision whether or not there is perfect agreement), you might get that by:

```
reference +.= ⍉sample
```

Permutation and partition matrices. You can permute or partition the members of one array by making one of the arguments of an inner product

a Boolean array. The Boolean matrix *b* might contain a column for each of the desired partitions. In the expression `b+.×ω`, a `1` in *b* indicates that a row of *ω* is included in a partition, and a `0` indicates that it is not.

When there are as many partitions as there are rows, and each row and column contains exactly one `1`, the matrix is a *permutation matrix.* It specifies an order in which the rows of *ω* are represented in the result of `b+.×ω`.

A Boolean array acts as a partitioning device when the verb ***g***, applied to `0` in the Boolean array and any value in the other array, yields the identity element for verb ***f***. For example, with the `+.×` inner product, `0×` any value in the other array yields `0`, which is the identity element for `+`. That is, elements marked `0` in the Boolean array have no effect on the sums.

Similarly, in an inner product such as `α×.*b`, raising an element of *α* to the power `0` yields `1`, which is the identity element for `×`. For other ways of partitioning, see also the section on the cut conjunction.

Analysis of connectivity. You can evaluate the connectivity of a Boolean graph with inner products. If each row of a matrix *c* indicates a node of a network, and each column a connection to that node indicated by a `1`, then the existence of a direct connection is found by

```
c ∨.∧ ⍉c
```

and the number of connections by

```
c +.∧ ⍉c
```

Applications in circuit analysis will be familiar to electrical engineers, but analogous calculations occur in other fields. For instance, the connectivity of an urban transit system might be documented by tabling the stops as rows of a matrix and the lines that serve them by columns. If such an array is called *map*, the connectivity of the system is found by:

```
c←(⍉map) ∨.∧ map
```

The result *c* is a square matrix showing where one line intersects another. A map showing which line is connected to another is found by

```
c ∨.∧ ⍉c
```

and showing which line can be reached in a two-step journey by

```
c ∨.∧ ⍉c ∨.∧ ⍉c
```

and so on. Continuing this process until the result does not change produces the *transitive closure* of the graph. In the case of the transit system above, it would show all stops you could get to from any other stop, and would tell you if indeed "You can't get there from here."

Other uses of `∨.∧` include the computation of cross-references, calling trees, and genealogy trees.

Inner product involving reduction by a non-associative verb. The dot conjunction's left argument ***f*** reduces each of the outer products generated with the right argument ***g***. The alternating sum or product may serve there as it does in reductions. For example, the MacLaurin series for approximation of the sine is shown conventionally as the alternating sum of ratios of successive odd powers:

$$sin\ x \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} + \cdots$$

In APL, this simplifies to the `-.÷` inner product, as follows:

```
sin α ⟷ (α∘.*odd) -.÷ !odd←¯1+2×⍳k ⊣ ⎕io←1
```

Monadic Use: The Alternant

The expressions `-.×ω` and `+.×ω` are, for square matrix arguments ω, the *determinant* and the *permanent* of mathematics.[8]

The generalization to arguments other than +, −, and × is based on construing the determinant as an alternating sum (`-≠`) over products over the diagonals of matrices obtained by permuting the major cells of ω.

Applied to a matrix of coefficients of a set of linear equations, the determinant identifies the character of the solutions; applied to the matrix of partial derivatives of a vector function on vector arguments it yields the volume transformation effected by the function; and applied to the matrix `s,1` it yields `!1↓⍴s` times the signed volume of the simplex of `n←1↑⍴s` vertices in `(n-1)`-space.

The determinant is generated by the expression

```
-.× M
```

where `M` is a rank-2 array, usually square, but possibly with more rows than columns. Only the "top-left square" portion of an array with more columns than rows is considered; that is, `M` is truncated to be `(⌊/⍴M)↑M`.

The general case ***f*** `.` ***g*** `M` is defined analogously as reduction by ***f*** over a set of `!n←1↑⍴M` "products" produced by reduction by ***g*** over `!n` distinct sets of `n` elements chosen one from each row and column of `M`.

[8] The discussion that follows is condensed from "A Dictionary of APL," and from K.E. Iverson, *Determinant-Like Functions Produced by the Dot Operator*, SATN 42, Toronto: I.P. Sharp Associates, 1982.

The ordinary determinant `-.×M` on a square matrix may be used as a test for invertibility before applying ⌹.

The determinant of a square matrix *M* may be defined as the alternating sum (`-/`) of the `!n←1↑⍴M` products over *n* elements chosen (in each of the `!n` possible ways) one from each row and column. Analogous calculations in which other function pairs are substituted for − and × lead to other useful functions; examples include the pairs `⌊.⌈M`, `∧.∨M`, and `+.×M`, the last (called the *permanent*) being useful in combinatorics.[9]

Suppose *M* is a magic square of order 3, then:

```
      M
6 7 2
1 5 9
8 3 4

      -.×M
360

      +.×M
900

      ∨.∧M
2
```

Non-Square Arguments

The extension of the alternant to non-square arguments allows the use of the determinant `-.×` on an *n*+1-by-*n* argument to compute the signed volume of the simplex it represents in n-space.

When *M* has more rows than columns, `×/s←(1↓⍴M)↑⍋⍳1↑⍴M` distinct sets of `1↓⍴M` elements can be selected without repetition of either column or row, and these are arranged in lexical order in an array of shape *s*. For example:

```
      ⊢M←?4 2⍴9
6 4
5 5
4 3
1 4

      -.×M
21
```

[9] See H.J. Ryser, *Combinatorial Mathematics*, Carus Mathematical Monograph Number 14, Mathematical Association of America, distributed by John Wiley and Sons, Inc., Providence, R.I., 1963.

For a wide matrix *M*, the result of ***f .g*** *M* is equivalent to ***f .g*** `(⌊\⍴M)↑M`.

The case of `-.×` applied to an *n*+1-by-*n* matrix is of special interest because it is equivalent to `-.×M,1`, and can therefore be used to compute the volume of the simplex defined by *M* without explicit catenation of a final column of 1s.

If *a*, *b*, and *c* are two-element vectors, then

```
      -.×M←3 2⍴a,b,c
```

yields twice the signed area of the triangle with vertices *a*, *b*, and *c*. The sign of the result depends on the order of the vertices, positive when they are in counterclockwise order and negative when they are in clockwise order. For example, if:

```
      a←1 1 ⊣ b←1 0 ⊣ c←0 1

      -.×3 2⍴a,b,c
¯1

      -.×3 2⍴b,a,c
1
```

The signed result of `-.×M` is significant in higher dimensions as well. If *a*, *b*, *c*, and *d* are points in three-space (in a dextral coordinate system), then

```
      -.× 4 3⍴a,b,c,d
```

gives `(!3)` times the signed volume of the tetrahedron, the results being zero if the points are coplanar, and positive if the points *b*, *c*, and *d* are in counterclockwise order when viewed from *a*. For example:

```
      a←1 1 1 ⊣ b←1 0 0 ⊣ c←0 1 0 ⊣ d←0 0 1

      -.×4 3⍴a,b,c,d
2
      -.×4 3⍴a,c,b,d
¯2
```

The use of functions other than − and × can be illustrated in the context of the so-called *assignment* problem. One of *n* persons (or machines) is to be assigned to each one of *n* tasks in some optimal way, the cost associated with assigning person *i* to task *j* being given by element `M[i;j]` of the square matrix *M*.

To minimize the sum of the costs in the assignment, the optimum value is given by `⌊.+M`, since the summation `+/` is applied to each of the `!1↑⍴M` possible assignments, and the minimum `⌊/` is applied over them.

Other useful function pairs in assignment type problems include `⌈.+M`, `⌊.×M`, `⌈.×M`, `⌊.⌈M`, and `⌈.⌊M`.

A square Boolean matrix `B` is said to be a *Latin square* if it contains exactly `1↑⍴B` ones, with one in each row and one in each column; one Boolean matrix `C` is said to *cover* another, `D`, if `∧/,C≥D`. The expression `∨.∧C` yields `1` if `C` covers a Latin square, and the expression `+.∧C` tells how many distinct Latin squares are covered by `C`.

8 *Verb Formation*

This chapter describes the various mechanisms by which you enter the definition of a user-defined verb or edit the definition once it has been entered. To write your definition appropriately, and to edit it when it runs into trouble, you will also need to be aware of the way a definition is used, how names are localized, what happens when an error or interrupt is encountered, and how such events can be handled automatically. These topics are covered in Chapter 9, "Control of Execution" and Chapter 10, "Event Handling"; you may need to consult those chapters in order to elucidate some of the points raised in this one.

Mechanisms for Defining and Editing Verbs

APL has three main syntactic classes: nouns and pronouns; verbs; adverbs and conjunctions. Members of these classes are formed and named in different ways.

- **A noun** is assigned a name by the *copula*, denoted by the left-pointing arrow ←. See Chapter 5, "Naming Nouns and Pronouns".
- **An adverb or conjunction** exists only as a primitive and may not be given a name. Currently, there is no mechanism for creating user-defined adverbs or conjunctions.
- **A user-defined verb** is produced by either of two routes:[1]
  - ***The definition editor.*** Because you invoke the editor by entering the symbol ∇, it is called the ∇-editor or del-editor.
  - ***The system verbs*** `⎕fx` ***and*** `⎕fd`. Each of these takes as argument an array of characters that represent the definition of the verb to be created. You can edit an existing verb by capturing its present definition with the verb `⎕cr` or `1` or `2 ⎕fd`, editing the resulting array, and then using `⎕fx` or `3 ⎕fd` to create a new definition that replaces the old.

  In addition to these ways of creating a new definition, a previously defined pronound or verb may be materialized in the workspace by:

  - ***Copying it from a saved workspace*** by means of the command `)copy` or `)pcopy`, described in Chapter 17, "System Commands". The copied verb creates or replaces the *global* referent

[1] Executing an adverb or conjunction also produces a verb. However, the *derived verb* thus produced cannot currently be assigned a name and thus exists only in passing, during the execution of a sentence that contains an adverb.

of the name. The `)copy` or `)pcopy` commands will not replace a verb that appears anywhere on the state indicator, and hence cannot be used to edit a verb that is in use.

- ***Defining it from within a package*** by means of the system verbs `⎕pdef` or `⎕ppdef`, described in Chapter 16, "System Nouns and Verbs". The verb thus defined creates or replaces the *local* referent of the name. `⎕pdef` and `⎕ppdef` will not materialize an object while an object of the same name is visible anywhere on the state indicator, and hence cannot be used to edit a verb that is in use.

Figure 8–1 summarizes the differences between the methods or defining verbs.

	∇-Editor	System verbs □fx or □fd
Invocation	***Interactive.*** You can invoke the editor from a T-task or an S-task (but not from an N-task or B-task), and only while your session is in *immediate execution* mode.	***APL primitive.*** Either system verb may be included in any APL sentence being executed, regardless of whether entered from the keyboard or part of a user-defined verb, and in any type of task (T-, S-, N-, or B-task).
Name Localization	***Global.*** The ∇-editor always creates or edits a verb having a *global* name and ignores any local use the name may have.	***Local.*** Either □fx or 3 □fd creates a verb whose name is at the *visible* (that is, most local) level. When the proposed name has not been localized, the most local level is the same as the global level; in that case, the ∇-editor and the verbs □fx and □cr affect the same object.
Editing During Suspension	***Suspended verbs.*** You can use the ∇-editor to edit a verb whose execution is *suspended* – that is, its name appears in the state indicator *with* an asterisk beside it. You *cannot* use the ∇-editor when the verb is *pendent* – that is, when its name appears anywhere in the state indicator *without* an asterisk.	***Any verbs.*** Either □fx or 3 □fd can redefine any verb provided the verb is visible. (However, it is possible to make certain changes that will prevent resumption of work on verbs that are pendent; see "Damage to a Pendent Verb," below.)

Figure 8–1 : Comparison of ∇-editor and □fx or □fd.

Representation of a User-Defined Verb

The *standard form* for display of a user-defined verb, used by the ∇-editor and returned by the system verb 1 □fd, marks a definition by the symbol ∇ at the beginning and end. Each line of the definition is assigned a number, shown at the beginning of each line in square brackets.[2] For example,

[2] The spacing on each line is generated by the system afresh each time a verb's definition is displayed, according to a standard convention. Because of that, redundant blanks that you may introduce when you enter the definition are discarded in the display.

here is the standard display of a definition for a user-defined verb called *growthrate*:

```
    ∇ z←time growthrate rate;r;t
[1]   r←0.01×rate÷12
[2]   →(2=⎕nc 'time')/dyad
[3]  monad:time←1
[4]  dyad:t←time×12
[5]   z←(1+r)∘.*t
    ∇
```

Each definition consists of a ***header*** (the top line) and a ***body*** (the rest).

Header The top line names the verb and illustrates its syntax. In the display, the top line has no line number; during editing, you refer to it as line `[0]`. By convention, the final ∇ appears on a line of its own.

Body The header is followed by a sequence of numbered lines called the *body*. The body shows the APL sentences the system should execute when the function is invoked. The SHARP APL system supplies the line numbers itself; it numbers the lines with consecutive integers starting at `[1]`.

Canonical Form

The *canonical representation* of a definition lacks the bracketed line numbers and the opening and closing ∇, but is otherwise the same. It is displayed flush-left:

```
z←time growthrate rate;r;t
r←0.01×rate÷12
→(2=⎕nc 'time')/dyad
monad:time←1
dyad:t←time×12
z←(1+r)∘.*t
```

The canonical form is returned (as a table) by the system verb ⎕*cr* or by 2 ⎕*fd* (see Chapter 16, "System Nouns and Verbs").

The system verb ⎕*fx* expects an argument in the canonical form. The system verb 3 ⎕*fd* accepts either the canonical form (as a table) or the standard form (as a list with embedded *newline* characters, with line numbers and opening and closing ∇).

Names in the Header

The header always contains the verb's name. This is the name by which the verb is invoked in an APL sentence. In the example, the verb's name is `growthrate`.

The header also indicates whether the verb takes arguments and returns a result. It does so by supplying names by which the argument and result are known inside the definition. When the verb takes *one* argument, a name for the argument follows the verb's name. In the example, `rate` is the way the verb refers to its right argument.

When the verb takes *two* arguments, names for the arguments appear one on either side of the verb's name. In the example, `time` and `rate` are names for the two arguments.

When the verb returns an explicit result, the header includes a name for the result. The result (if present) must be the first name in the header, and must be followed by the copula. In the example, `z` is a name for the result. When execution of the verb terminates, the result returned is whatever value has been assigned to the name shown in the header as the result.

If the header does not provide for a result, or execution terminates without assigning a value to the name that the header designates for the result, the verb can still be executed. However, when the sentence that invokes the verb tries to make use of the verb's (nonexistent) result, SHARP APL reports a `result error`. Thus, a verb that returns no result can be executed only when it is the sentence's root verb.[3]

Headers for Monadic, Dyadic, and Ambivalent Verbs

A *monad* is a verb that takes one argument. The header of a verb that can be used only as a monad therefore provides a name for only one argument. The name for the argument *follows* the name of the verb.

An *ambivalent* verb can be used either as a monad or as a dyad. A user-defined verb whose header provides names for two arguments is ambivalent, and may be used with one argument or with two. When you use the verb dyadically, you must place its arguments around it, one on each side of the verb. Inside the definition, the values supplied as arguments are referred to by the names shown in the header.

When you use an ambivalent verb monadically, you place the argument *after* the verb. Immediately to the verb's left, there must be no noun that

[3] For discussion of the root verb, see the sections on order of execution in Chapter 3, "Grammar".

could be its left argument.[4] Inside the definition, the value you supplied for the right argument is known by whatever name appears in the header to the right of the verb's name. The name to the left of the verb's name is reserved for the left argument. During monadic use, that name has no value. A verb intended for ambivalent use should therefore test for the existence of the left argument (as, for example, in line 2 of `growthrate`).

Localization of Names

Except for the name of the verb itself, every name that appears in the header is thereby declared to be a *local* name. In particular, the names for the verb's arguments and result are local names. *Inside the definition,* those names are used to refer to the nouns supplied as the verb's arguments.

The names used as *labels* for lines within the definition are also local to the definition. (In the example, `monad` is a label for line 3, and `dyad` is a label for line 4.)

A label is a name that occurs at the beginning of a line and is separated from the rest of the line by a colon. While the verb is being executed, the name has a value equal to the number of the line to which it is attached. Thus, during execution of `growthrate`, `monad` has the value `3` and `dyad` has the value `4`. Although a label can be *used* in the same way as any other noun (for example, to compute `monad+1`), its value is fixed; you cannot assign a new value to a label.

Additional local names may be declared by listing them in the header, to the right of the names for the verb and its arguments and set off by semicolons. In the example, the names `r` and `t` are localized. This means that, during execution of `growthrate`, the only visible uses of those names are the ones assigned by `growthrate`. After execution of `growthrate` is complete, all local meanings of those names disappear and their prior (more global) meanings – if any – are again visible.[5]

Using the ∇-Editor

When you are in immediate execution mode, entering a line whose first non-blank character is ∇ switches you to definition mode. You remain in definition mode until you enter a line whose last non-blank character is ∇.

[4] A noun that is to the verb's left but is the argument of a conjunction does not count. For example, in the phrase `a+⍤2 ambiverb x`, the number 2 is not available to be the left argument of `ambiverb`, and so the verb is here used as a monad.

[5] The name of the verb itself is *not* local. You could make it local by including it in the list in the header or using it as a label. When you make the verb's own name local to it, you make it impossible to invoke the verb again until execution from the first invocation is complete.

(A ∇ embedded in a phrase or inside a character constant or a comment does not count.)

What you enter with the ∇ depends on whether you are defining a *new* verb or editing an existing verb definition.

New definition `∇ z←myverb price;subtotal`
Enter ∇ followed (on the same line) by the verb's header. The header *must* include the verb's name. It should also include a name for the verb's result, its arguments, and its local names (as illustrated above or in the example for `growthrate`). If you are not sure of some of these names, you can put in just the ones you already know and use the editor to supply the other names later.

Existing definition `∇ myverb`
Enter ∇ followed by the *name* of the verb (with no mention of its result, arguments, local names, etc.).

Prompts in the ∇-Editor

The editor prompts by entering the number of the next line (in brackets). When you are just starting a new definition, the "next" line is line 1; so for a new definition the initial prompt is `[1]`. For a definition that already has three lines, the initial prompt is `[4]`.

To the right of the prompt, the editor waits for you to enter the text you want on that line, or an editing command. An editing command starts with `[` or `)`. A line of text whose first non-blank character is something other than `[`, `)`, or ∇ becomes the definition of the line whose number the system displayed at the left.

If you want to change the line number, use can either edit it just like any other part of the line, or enter a new line number after the existing line number prompt.

Editing Commands

`[`***n***`]` Define (or redefine) line ***n***.

`[Δ` ***n1 n2*** `...]` Delete lines ***n1, n2,*** ...

`[□]` Display the entire definition.

`[`***n***`□]` Display line ***n***.

`[□`***n***`]` Display from line ***n*** to the end.

`)` Copy the last immediate-execution line as the body for the current line, leaving the cursor at the end of the line.

`)`***p*** Copy the last immediate-execution line as the body of the current line, and go into two-pass line editing with the cursor initially at position ***p*** (as described for `[`***n*****□*****p***`]`, above).

`[`***n*****□*****p***`]` Edit line ***n*** at position ***p***.[6]

p=0 One-pass editing (best for screen terminals). Leaves the cursor at the end of the line. You may backspace and (if the device permits) overwrite or insert corrections.

p≠0 Two-pass editing (for hard-copy terminals). Leaves the cursor at position ***p*** of the line below. You may then enter editing commands as follows:

To the left of the first dot or comma:

`/` deletes the character above

n inserts ***n*** blanks before the position at which you enter ***n***. Entering a letter inserts 5 times the letter's position in the alphabet. For example, `a` for 5 blanks, `c` for 15 blanks, etc.

Any characters to the right of the first dot or comma:

The text is inserted in the line above, to precede the character above the dot.

For example, the commands in Figure 8–2 might be used to replace the name `defaultinterval` by `time`.

When what you enter contains:

`,` The editor amends the line as you have indicated, inserting the text typed after the comma at the position indicated by the comma, displays the amended line, and awaits further two-pass editing instructions.

`.` The editor amends the line as for comma, but does not display it. This completes two-pass editing.

[6] The procedure for line editing within the ∇-editor are essentially the same as those for line editing in immediate execution; see the discussion of the system command) in Chapter 17, "System Commands".

/ or a digit. The editor amends the line by deleting positions marked by a slash, and inserting ***n*** spaces in front of each position marked by ***n***.

The editor positions the cursor at the first of the inserted blanks, or at the end (when you did not insert any blanks). You may enter a correction by overstriking the blanks (or any of the other characters, provided you form legal overstrikes). When you have revised the line and press *Enter*, this completes two-pass editing.

) Copy the last immediate-execution line as the body for the current line, leaving the cursor at the end of the line.

)***p*** Copy the last immediate-execution line as the body of the current line, and go into two-pass line editing with the cursor initially at position ***p*** (as described for `[`***n***`□`***p***`]`, above).

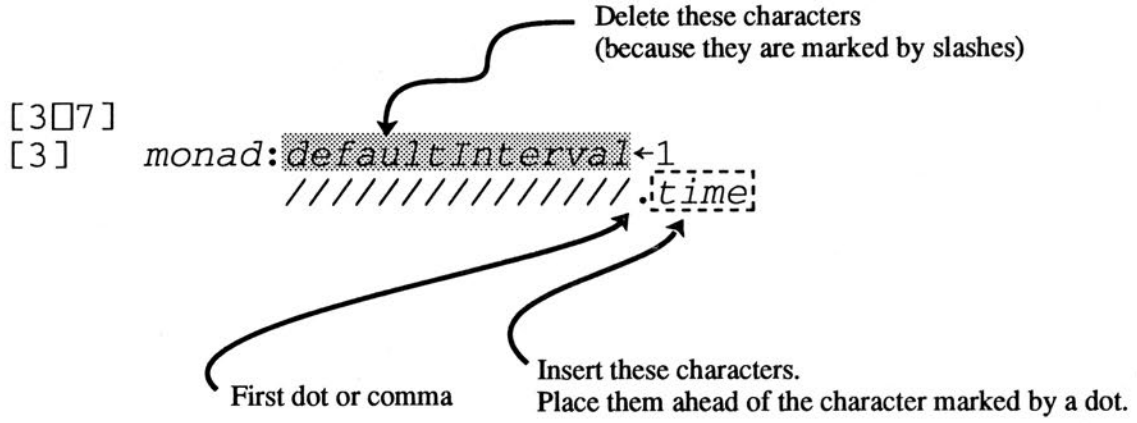


Figure 8–2 : A typical line edit using the ∇-editor.

Interpolating Lines Between Existing Lines

You interpolate a line by giving it an interpolated line number. For example, to insert a new line between lines 3 and 4, enter `[3.1]` (or any other fraction greater than 3 and less than 4).

While you continue editing, the editor leaves the lines with whatever numbers you assign them. But when you leave the editor (thereby closing the definition), the editor renumbers all lines with consecutive integers.

Once you have entered a line, the editor prompts for the next line in sequence. It does that by adding 1 to the last position used in the line just completed. When the previous line had an integer line number, the editor's next prompt is 1 more than that. When the previous line was a fraction with two decimal places (for example, `[6.02]`), the editor adds .01, so its

next prompt is `[6.03]`. It may prompt for an existing line number, so be careful not to enter an inserted line over an extant one.

Empty Lines

Empty (all blank) lines cannot be entered using the editor. Although empty lines are accepted within APL functions, they can only be entered using the system verbs ⎕*fx* and 3 ⎕*fd*. The editor will, however, allow display or deletion of empty lines.

Syntactically Invalid Lines

Lines which are syntactically invalid are accepted by both the editor, and by the system verbs ⎕*fx* and 3 ⎕*fd*. Syntactically invalid lines include those which contain unbalanced quotes, third alphabet characters not in quoted strings, invalid numeric constants, and invalid system noun or system verb names.

Box-drawing Characters

The system verbs ⎕*fx* and 3 ⎕*fd* allow you to enter lines which contain the box-drawing characters for IBM 327x-type terminals (⎕*av*`[241+⍳11]`) in quoted strings. The ∇-editor will allow you to display or delete such lines, but does not permit their entry.

Editing Occurs in Real Time

You cannot "quit" the ∇-editor, as you usually can do with text editors. Each line of the verb is altered as you enter that line. If you have made an error in editing a verb, the best approach is to obtain a fresh copy of the verb (perhaps by)*copy* from a saved copy of the workspace) and start over.

You can, however, abandon editing of a single line being edited with apl [***n*⎕*p***], by entering an input interrupt sequence, such as *o*-backspace-*u*-backspace-*t*.

Closing the Definition

When you are satisfied with what you have entered, you enter ∇ to indicate that editing is complete. The character may stand alone, or it may appear at the right end of a line you have just entered.[7] It does not matter what line you are working on when you enter the closing ∇; the ∇ does not mean "This line is the end," but rather "I have finished editing."

[7] Provided the ∇ is not part of a comment or part of a constant (that is, enclosed in quotes).

The editor renumbers the lines with consecutive integers and creates the newly defined verb. There is no specific acknowledgment, other than the fact that the APL system returns to immediate execution mode and sends the immediate-execution prompt (six blanks).

When you edit a previously existing verb and change its name by editing line 0, closing the definition both creates the new definition and erases the old.

Locking the Definition

If you exit from definition mode with ⍫ rather than ∇, the definition is both closed and locked. The editor rejects an attempt to use the ∇-editor on a locked definition with the message `defn error`. The system verbs `⎕cr` or `⎕fd` simply return an empty array as the definition of a locked verb. Furthermore, the text of the line will not be displayed in error reports nor will it appear in `⎕er`.

The SHARP APL system provides no means to unlock a locked verb once you have locked it. It is prudent to keep somewhere safe an unlocked version of any verb you lock.

Editing Can Prevent Resumption of Execution

One of the very useful features of an APL system is that when a user-defined verb encounters an error, execution halts but is not abandoned. You can correct the definition and resume execution without having to abandon the work already done. (see Chapter 9, "Control of Execution".)

However, editing the definition of verb while it is being executed (that is, editing a verb whose name appears on the state indicator) has some risks. It is possible to modify the definition in such a way that you cannot resume execution of the verb you edited, or so that it is no longer possible to return to the verb that invoked it. Problems created by editing a verb whose execution has not terminated are lumped together under the name *state indicator damage*. Attempting to resume execution of a verb whose definition has been rendered inaccessible or ambiguous is rejected as event 9, `si error`. This event can be trapped, but recovery may require cutting back the stack (abandoning the more recent work) and resuming from an earlier point. (see Chapter 10, "Event Handling".)

Distinguishing Active, Pendent, and Suspended Verbs

In discussing the effects produced by changing the definitions of verbs before they have finished, it is useful to distinguish three situations. A verb may be *active*, *pendent*, or *suspended*. Here is what these terms mean:

Active The condition of the user-defined verb now being executed. Its name is at the top of the execution stack. The state indicator (reported by `2 ⎕ws 2`) shows no asterisk beside its name because it is not halted.

This is a situation you can never see from immediate execution. Since the ∇-editor can be invoked only from immediate execution, it can never edit the active verb. However, `⎕fx` and `3 ⎕fd` *are* permitted to edit the active verb.

Pendent The condition of a user-defined verb that has invoked another user-defined verb. The pendent verb is waiting for the verb it invoked to complete execution. When that happens, the pendent verb automatically becomes active again. Until then, it has nothing to do. It is waiting in mid-line (since it has started execution of a line in its definition but has not yet completed that line).

Suspended The condition of a verb whose execution has been halted because of an error or interrupt. A suspended verb is marked in the state indicator by an asterisk.

When a verb's execution is interrupted, the system looks for a `⎕trap` expression or `⎕ec` to tell it what to do next. If it does not find a trap that covers the event, it displays a message describing the event and reverts to immediate execution mode. Now if you execute the command `)si`, you will see the verb's name at the top of the stack with an asterisk beside it.

Damage to the Active Verb

It is possible for the active verb to modify itself by a statement that uses `⎕fx` or `3 ⎕fd`. When redefinition changes the line being executed, SHARP APL reports an `si error`. A change is considered to have occurred when, following use of `⎕fx` or `3 ⎕fd`, the line whose number is reported in the first item of `⎕lc` is no longer identical to the line the interpreter started with.[8] That could arise either because:

- you actually changed that line

  or

- you inserted or deleted lines ahead of it so that the first element of the line counter now points to a line with different content.

[8] The comparison is based on an internal representation formed during analysis of the line for execution and not on the characters in its representation. It is possible for lines that contain the same characters nevertheless to differ. It is also possible for lines that differ only in redundant blanks to be accepted as identical.

When SHARP APL detects such a change to the current line of the active verb, it replaces the verb's name in the state indicator by blanks and signals event 9, `si error`.

Damage to a Pendent Verb

The name of a pendent verb appears in the state indicator without an asterisk. (The active verb is a special case of a pendent verb.)

The active verb's use of `⎕fx` or `3 ⎕fd` may damage a pendent verb in the same way as it may damage the active verb itself: by changing the sentence to which the line counter points. When the active verb completes execution and attempts to return control to the pendent verb, the interpreter signals event 9, `si error`.

SHARP APL may be unable to resume execution of a pendent verb if you change the localization of its name in a way that makes the verb invisible. That can happen in either of two ways:

- *You localize the verb's name.* That is, you edit a verb whose name appears lower on the state indicator so that the pendent verb's name becomes local to it.[9] The pendent verb is still there, but you can no longer see it.
- *The verb's name is initially local, but you delocalize it.* That is, the pendent verb's name is local to a verb lower down on the state indicator. You revise the definition of the lower verb so that the name is no longer local. The local meaning vanishes.

In either case, SHARP APL replaces the verb's name in the state indicator by blanks. It leaves `⎕lc` unchanged. When the pendent verb completes execution and tries to return control to the verb that invoked it, it finds the name is now blank. It signals event 9, `si error`.

When work is suspended and the top row of the state indicator contains a verb whose name is blank, a branch statement in the recovery expression of a relevant `⎕trap` (or a branch statement entered from the keyboard) causes a normal exit (but with no result). That line is removed from the state indicator, and control passes to the line that followed it. That line may be able to resume – provided it is not expecting a result.

[9] Remember that labels and the names for the verb's arguments and result are also local names.

9 *Control of Execution*

This chapter describes how you "run" a program – that is, start the execution of a sentence containing a user-defined verb. It describes what happens as that verb uses other user-defined verbs and how they refer to nouns to pass data between them. It also describes what happens when the program encounters an error or is interrupted, how you can make changes or corrections to the suspended program, and how you can resume work without having to start over from the beginning.

Programming and Immediate Execution

In its default state, SHARP APL is an interactive system. That is, when you start an APL session from a keyboard, the SHARP APL system waits for you to type one sentence from the keyboard. When you signal that your sentence is complete (by pressing the *Enter* key), the SHARP APL system interprets and executes that sentence and displays its result. Then it waits for you to type another.

If each of the sentences you type is a small calculation, you get the answer immediately; the session is a rapid alternation between question and answer.

However, the sentence you type may invoke a user-defined verb (or several of them, in a compound sentence). The verb's definition may contain up to 32767 lines. Each of the sentences on those lines may itself invoke other user-defined verbs. They in turn may invoke others. Thus a single sentence you enter from the keyboard may set in motion a great deal of work.

A verb's definition consists of a sequence of APL sentences. In principle, anything a program does you could also do by entering instructions from the keyboard. You could simply enter each sentence, watch it execute, and then enter the next.[1] A verb's definition thus consists of a set of sentences that you *might* have entered from the keyboard for immediate execution. Instead, you – or someone – entered them earlier and made them part of a definition. Perhaps you used the ∇-editor to accumulate them, or perhaps a word processor. Perhaps you acquired the whole set of definitions simply by entering the command `)load` followed by the name of a saved workspace

[1] There are a few things that occur only within the environment of a user-defined verb and thus cannot be done solely by entering instructions from the keyboard for immediate execution. For instance, the localization of names is a property of a user-defined verb. The *environment condition* `⎕ec` assumes a default value when made local to a user-defined verb. These could be manipulated by expressions typed from the keyboard, but only while a user-defined verb is suspended. See the discussion of `⎕ec` in Chapter 10, "Event Handling".

and thereby gained access to any of the definitions stored in it. Perhaps you have a `⎕trap` set up so that each time you or your program asks for something that is not now defined, it retrieves what is needed from a file.

Once you have the definitions for a program, you instruct SHARP APL to execute the program simply by including its name in a sentence to be executed.

Status of Work in Progress

Each workspace contains a *state indicator* and a *line counter.* Together, these show the name of the verb now active and the line of that verb now being executed. They also show the verb that invoked the active verb, and the verb that invoked it – for as many verbs as are involved. The state indicator contains a row for each defined verb whose execution has started but not yet completed. It also shows the symbols ⍎ and ⎕ in the same manner as user-defined verbs.

When no user-defined verb is pending, the state indicator is empty. When you enter a sentence containing a user-defined verb, the name of the verb to be executed goes on the state indicator. At the same time, the line counter keeps track of the number of the line that is now being executed (or that is ready to execute, when the interpreter is between lines).

If, before work on that verb is complete, its definition invokes another user-defined verb, that verb's name is added as a new row at the top of the state indicator. SHARP APL sets aside work on the row below and focuses on the row at the top. When it finishes work on that top row, it removes it from the state indicator stack and resumes work on the row that was temporarily pushed down.

The command `)si` (*state indicator*) shows the names that are on the state indicator stack, commonly just called the *stack.* You can only enter a system command when your task is in immediate execution mode. Thus the command `)si` has something to show you only when work has been interrupted, leaving the names of some user-defined verbs on the stack. The system verb `2 ⎕ws 2` reports the same information. However, since `2 ⎕ws 2` does not require a return to immediate execution, it can report what is on the stack as it goes along. To illustrate, consider the following two programs:

```
    ∇ z←verb1
[1]   'This is verb 1, and here is si:'
[2]   2 ⎕ws 2
[3]   z←⍴⎕lc
[4]   z←z,verb2
    ∇
```

```
    ∇ z←verb2
[1]   'This is verb 2, and here is si:'
[2]   2 ⎕ws 2
[3]   z←⍴⎕lc
    ∇
```

When you execute *verb*1, you see first the state indicator as it was when line 2 of *verb*1 was active, then the state indicator as it was during the corresponding line of *verb*2, and finally the result:

```
    verb1
This is verb1, and here is si:
verb1[2]
This is verb2, and here is si:
verb2[2]
verb1[4]
1 2
```

The result `1 2` is the number of items on the state indicator during execution of *verb*1 and *verb*2.

Suspension Due to Error or Interrupt

Something may happen that prevents completion of the execution of a verb. Something in the environment may be unsatisfactory (a file is not available, or there is not enough space in the workspace). Something needed may be missing (a utility verb may be invoked but not present). The definition itself may contain an error of spelling or syntax.

Work may also be suspended by a signal from outside. You can signal an interrupt at two levels of urgency:

Weak interrupt Signaled by a single press of the *PA1* or *Break* key. Requests SHARP APL to halt before beginning the next line of a user-defined verb.

Strong interrupt Repeating the signal upgrades a former weak interrupt on which SHARP APL has not yet acted. It becomes a strong interrupt. It requests a halt to processing as soon as possible, without waiting for the start of the next line, or even for completion of the current line.

In general, after encountering an error or receiving an interrupt signal, SHARP APL stops working on the sentence that is now active. The interpreter checks to see whether the condition is provided for in an *event trap* (described in the next chapter). If no trap is provided, SHARP APL halts, *but does not abandon execution.* It notes where it is in the active

program. Then it returns to immediate execution mode and awaits further instructions.

You are free to take corrective action and then resume work. Or you may do something else for a while and then take corrective action and resume work. Or you may abandon execution of the most recent keyboard entry by entering → alone, or abandon *all* executions by entering `)sic` (*state indicator clear*).

When you save your active workspace, the line counter and state indicator are saved as part of it. When the saved workspace is loaded at some later occasion, it is recovered as it was, including the state indicator showing where it is in the verbs you left partially executed. If it is appropriate, you will be able to resume execution where it was suspended.[2]

Pendent vs. Suspended Verbs

A verb whose execution has stopped because of an interruption or an error is said to be *suspended.*

A verb whose name *is* in the state indicator but is *not* suspended or active is said to be *pendent.* Although it is not currently active, nothing is wrong with it; it is simply waiting until execution of the verb that it invoked is completed. Execution of a pendent verb resumes automatically as soon as execution of the verb above it in the state indicator is completed.

The result of `2 ⎕ws 2` (or the display produced by `)si`) shows on each row the name of a verb, followed in brackets by the number of the current line for that verb. A suspended verb is marked by an asterisk, whereas a pendent verb has no asterisk.

The Line Counter

The system noun `⎕lc` returns a numeric vector having one element for each row of the state indicator; it contains in numeric form the line numbers shown in brackets in the display of the state indicator.

For a pendent verb, the line number shown is the number of the line now being executed. For a suspended verb whose execution was halted by an error or interrupt, `⎕lc` shows the number of the line the system was executing when the error or interrupt occurred. For a verb halted after execution of one line has been completed, but before execution of the next line has started, `⎕lc` shows the number of the line next to be executed if execution is resumed in sequence.

[2] However, saving a workspace does not record what files were in use ("tied"), and loading a saved workspace does not automatically re-tie them. If the sentence being executed requires use of files, you have to make provision to record what they were and to resume their use when the workspace is loaded. Similar restrictions apply to the use of shared variables. See Chapter 14, "APL Component Files" and Chapter 11, "Shared Variables".

When the system puts the symbol ⍎ or ⍞ on the state indicator, the corresponding element of `⎕lc` contains the line number of the verb that contains the ⍎ or ⍞. When you enter ⍎ or ⍞ from the keyboard (in immediate execution), the corresponding element of `⎕lc` contains the line number of the most recently suspended verb. However, for ⍎ when no user-defined verb is on the state indicator, `⎕lc` contains `0`.

Local Names at Each Level of the State Indicator

Each user-defined verb may declare that certain names are *local* to its definition. The local names are those used in the header to name its arguments and its result, and those listed further to the right, set off by semicolons. A verb's *line labels* are also local.

However, the name of the verb itself is not local unless that name is explicitly repeated in the list of local names, or is also the name of a label, an argument, or the result.

As long as a verb works satisfactorily, you never need to know what names within it represent its arguments or its result, or what other names it may localize. These become significant *only* when you halt the verb or change its definition.

The command `)sinl` (*state indicator with name list*) shows the same information as `)si`, but lists on each row the names that are localized at that level.

To illustrate the effects of localization, suppose that there exist in the workspace three objects called `a`, `b`, and `c`. Suppose that you invoke a verb named `verb1`, whose line 4 invokes `verb2`, whose line 3 invokes `verb3`, whose line 7 invokes `verb4`. Suppose that `verb4` refers to all three names, `a`, `b`, and `c`.

The definitions are written so that `verb1` and `verb4` each localizes the name `c`, while `verb3` localizes the name `b`. If you halt execution of `verb4` (while it is working on line 2) and display `)sinl`, it looks like this:

```
      )sinl
verb4[2] * c
verb3[7] b
verb2[3]
verb1[4] c
```

To make clear how localization determines what referent is visible, Figure 9–1 arranges all names of interest so they are aligned in columns. What is visible at a particular instant is the view looking downward from the top row of the display produced by `)sinl`. For this purpose, the "global" level

is a row below the last row of `)sinl`, containing all names that are visible when *no* verb is being executed.[3]

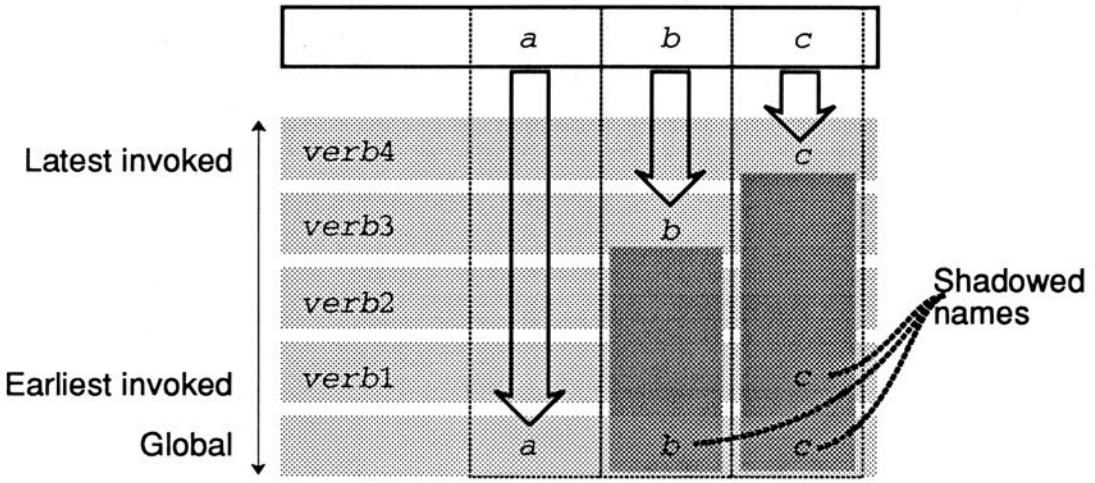


Figure 9–1 : The visible referent of a name is the first appearing in `)sinl`.

As you can see in Figure 9–1, since the name `a` is not localized by any of the verbs, a reference to `a` in `verb4` goes all the way down to the global level.

Since the name `b` is localized by `verb3`, a reference to `b` in `verb4` gets the `b` belonging to `verb3`. The local use of `b` is said to *shadow* the global `b`.

Since the name `c` is localized by `verb4`, a reference to `c` in `verb4` gets the `c` belonging to `verb4`, which shadows both the local use in `verb1` and the global `c`.

The system verb `5 ⎕ws` reports similar information. However, it reports it as a table in which there is a *row* for each name and a *column* for each level of the state indicator. You give it a list of names, and it reports the status of each name at each level, plus an extra column for global status. (See the description of `⎕ws` in Chapter 16, "System Nouns and Verbs".)

Dynamic Localization

The localization rule described in the preceding section is known as *dynamic localization*. That is, the names that a verb localizes appear global to the verbs it invokes.

[3] Strictly speaking, the bottom row of the figure should also show columns for the names `verb1`, `verb2`, etc., since those names are visible when no verbs are being executed.

Steps at the Start of Execution of a User-Defined Verb

Arguments are passed to a verb, and results returned, *by value*. A verb's argument is whatever noun is produced by evaluating the expression to the right or left of it. When the system starts to execute a defined verb, the following things happen:

- SHARP APL puts the verb's name at the top of the state indicator.
- SHARP APL appends a new item to the front of `⎕lc`, indicating which line in this verb is next to be executed. Initially, this number is `1`. Later, as execution of the verb progresses, the number in `⎕lc`'s first item is revised so that it always shows the line now being processed or ready to be processed next. The first member of `⎕lc` always refers to the verb most recently started.
- SHARP APL prepares names for the values of the arguments. It does not matter by what names the arguments were known outside the verb (or even whether they had names at all). The names of the arguments are local. So are any other names mentioned in the header.
- SHARP APL prepares a name corresponding to each label and gives it a numeric value indicating the line to which it is attached. A label differs from other local variables in that:
  - it gets its value automatically at the start of verb execution
  - its value cannot be changed.
- All other names appearing in the verb header are local. That is, they have no referent until assigned one during execution of the verb.

Any object outside the verb, but having the same name as an argument, result, label, or local name of the verb, is invisible while the verb is being executed and is unaffected by any action performed while the verb is active.

SHARP APL executes the lines of the verb in sequence (or as branch statements may direct) until the verb ends. At that point, the value of whatever variable was named in the header as the result of the verb is returned. The first element of `⎕lc` is dropped and the verb's name is removed from the state indicator. The referents of all names local to the verb vanish. The earlier (more global) referents of those names (if any) become visible once more. If a previous line on the state indicator indicated a pendent verb, it would become active again.

Example of Argument and Result Passing

Chapter 8, "Verb Formation" showed a verb `growthrate` to illustrate the standard form of a definition. The left argument was the number of years for which monthly compounding continues, and the right argument the annual percentage rate of interest. In the definition, the left argument was called

time and the right argument *rate*. Suppose a verb *report* contains on line 3 the sentence:

```
Total←Inv×Years growthrate BaseRate←0.5×⍳6
```

As SHARP APL starts on this sentence, the state indicator contains:

```
report[3]
```

In processing the sentence, SHARP APL finds that the verb *growthrate* is a dyad. When SHARP APL has found the value of *Years* and the value of *BaseRate*←0.5×⍳6, it invokes the verb *growthrate*. It leaves *report*[3] pendent while it puts a new verb at the top of the state indicator.

If at that point you could see the state indicator with name list, it would appear thus:

```
growthrate[1] z rate time dyad monad
report[3] z Total Inv Years BaseRate
```

As soon as *growthrate* is placed on the top of the state indicator, SHARP APL reserves the local names you can see beside *growthrate*[1]. The labels *monad* and *dyad* are given the values 3 and 4 respectively.

The right argument *rate* receives as its value the array that results from evaluating *BaseRate*←0.5×⍳6.

The left argument *time* is assigned the value of the array *Years*.

The name *z* is given no value; it is reserved for the result.

Multiple Copies of Arrays Share the Same Space

In this example, when *growthrate* is invoked, its left argument is not an expression but simply the name *Years*. Since *Years* already exists and the new name *time* is to have the same value, SHARP APL simply notes that *time* is the same as *Years* and does not create a second copy of the data. However, if the definition of *growthrate* subsequently sets a new value for *time* (for example, by a sentence such as *time*←*time*×12), SHARP APL then gives *time* its own value and no longer refers to the value of *Years*.

In general, whenever a name is assigned a value already identified by another name (for example, in an expression such as *x*←*y*←*z*←⍳*n*), SHARP APL creates only a single data array and lets all the other names refer to it, rather than making many copies of the same array. As soon as there is any change to any part of the value assigned to any of the names, SHARP APL creates a separate copy of the entire object for that particular name. This aspect of internal housekeeping is not directly visible other than by its

benefits in reducing the requirements for storage in the active workspace, and a reduction in the time required to perform the operation.

Many expressions in SHARP APL involving verbs and derived verbs save time and space in this way when their result is the left or right argument. Such expressions are called *identities*. For example, ravel of a list does no actual data movement. It takes fixed time and space to execute, regardless of the size of the argument.

Value Returned by the Verb at Completion

When execution of `growthrate` is complete, the name `growthrate` is removed from the top of the state indicator and SHARP APL picks up where it left off in its evaluation of line 3 of `report`. As `growthrate` terminates, SHARP APL notes the name assigned to its result (in this case, `z`). Whatever value `z` then has is returned to the statement that invoked `growthrate`. Thus, in the example, whatever value `z` then had becomes the value for the right argument of ×.

Branch Controls the Sequence in Which Lines Are Executed

Unless a branch statement (see next section) modifies the order, SHARP APL executes the lines in a verb's definition one after another in ascending numeric order starting with line 1.

A sentence whose first non-blank character, not counting a label and its colon, is the right-pointing arrow →, is a *branch* statement.

A branch statement overrides the normal sequence of execution by telling SHARP APL the number of the line to execute next. A branch can also be used to resume suspended execution, either manually from the keyboard or as part of an automatic recovery set up by `⎕trap` (described in Chapter 10, "Event Handling"). A suspended verb can be restarted only at the beginning of a line (even though the interrupting event may have occurred in mid-line).

A branch affects *only* the verb at the top of the state indicator.

When execution is suspended, entering a branch statement from the keyboard causes the verb at the top of the state indicator to resume execution at the beginning of the line indicated in the branch statement.

Form of a Branch Statement

The branch arrow → may stand alone or may be followed by an expression. The expression to the right of the arrow is the *destination* of the branch. When evaluated, the destination must be an item or a list. When the list is

not empty, its *first* element must be an integer. SHARP APL rejects any other value as event 11, *domain error*.

The expression to the right of the arrow is usually stated as some function of one or more of the verb's line labels, or 0 (meaning "exit"). However, the statement is not *required* to refer to a label. A label is a handy way of referring to a line's number. Labels also improve the maintainability of APL programs.

Result and Effect of a Branch

Branch is not a verb and does not have a result.

When the destination of a branch identifies an existing line, control passes immediately to that line. When the destination is a list having more than one item, items after the first are ignored.

Effect When Destination Is Zero or Out of Range

A destination that is an integer but is *not* the number of any line in the current verb (typically, 0) forces a normal end to the verb's execution. The verb returns whatever value has by then been assigned to the name that appears to the left of ← in the header.[4]

When the exit occurs before the result has been given a value, the verb returns no result. If the sentence that invoked the verb needs the verb's result, the failure to return a result causes event 8, *result error*.

Effect When Destination Is Empty

During execution of a user-defined verb, branch to an empty destination has no effect. The empty branch may arise directly from

```
→⍳0
```

or (more likely) from

```
→(false condition)/label
```

Following an empty branch, execution continues in sequence with the next sentence. (On a line with diamonds, that is the next sentence to the right. Otherwise, execution proceeds to the line below.)

When execution is halted, an empty branch does not restart it.

[4] The system verb □*signal* also terminates execution of the active verb, but by forcing an *abnormal* end. The event identified in □*signal*'s argument is reported to the verb that invoked the active verb. See the description of □*signal* in Chapter 16, "System Nouns and Verbs" and the discussion of event handling in Chapter 10, "Event Handling".

Branch in Execute

The foregoing statements about branch remain true regardless of whether it occurs within the argument of ⍎ (*execute*). Thus

```
⍎'→dest'
```

has exactly the same effect as

```
→dest
```

Branch in a Line with Diamonds

The foregoing statements about branch remain true regardless of where the branch statement is located within the line. For example, suppose a line contains three sentences separated by diamonds and a branch in the middle sentence, thus:

```
sentence1 ⋄ →elsewhere ⋄ sentence2
```

When the shape of `elsewhere` is empty, both `sentence1` and `sentence2` are executed. But when `elsewhere` is not empty, `sentence1` is executed and `sentence2` is not.

Branch to Resume Suspended Execution

When a verb has been suspended, the system waits for you to indicate what you want done with it. The system does *not* abandon the suspended verb. You do not have to resume execution at once; you are free to start execution of any verb whose definition is visible. (If you do so, those new verb calls appear at the top of the state indicator, above the entries showing the suspended verb.)

When execution of a user-defined verb is suspended (so that the suspended verb's name is at the top of the state indicator and has an asterisk beside it), a branch statement with a non-empty destination causes the active verb to resume execution at the line that is the branch's destination. The branch statement that resumes work:

- may be entered manually from the keyboard

  or

- may be part of the recovery expression of a trap.[5]

Generally, when execution has been halted, the expression `→⎕lc` resumes work. Note, however, the effects of a mid-line halt, discussed next.

[5] See Chapter 10, "Event Handling".

Resumption Following a Mid-line Halt

When the halt occurred between lines (for example, in response to a weak interrupt signaled from the keyboard) →⎕`lc` causes execution to continue with the next line in sequence.

When the halt occurred in mid-line (for example, in response to an error or a strong interrupt), ⎕`lc` contains the number of the line on which SHARP APL was working. Part of that line may have been executed already. Execution always resumes at the *beginning* of a line. Therefore →⎕`lc` resumes at the *start* of the line, even though part of it was already executed before the error or interrupt. The caret included with the error display (or in ⎕`er`) indicates how much of the line has already been attempted. Depending on what the line contains, repeating it from the beginning may be harmless or may be undesirable, leading to incorrect results. A line written so that restarting it from the beginning is always harmless is said to be *restartable.*

Naked Branch to Abandon Execution

A *naked branch* is a sentence in which the right-pointing arrow stands alone, with nothing to right of it, thus:

```
→
```

Execution of the active verb (that is, the verb at the top of the state indicator, regardless of whether it is running or suspended) is aborted, *along with the entire sequence of verbs that invoked it.*

SHARP APL removes from the state indicator the current top row (containing the name of the active verb) and any other verbs down to (but not including) the next verb marked with an asterisk.

Clearing the Entire State Indicator

The system command)`sic` completely clears the state indicator. Execution of all verbs on the state indicator is aborted.[6]

Branch During ⎕-Input

A naked branch within ⎕-input acts in the same way as a naked branch during execution. That is, the evaluation of ⎕ is aborted and so is the entire sequence of verbs that invoked ⎕.

[6] This command was formerly called)`reset`.

Branch with a destination is not permitted during ⎕-input. If you attempt it, the system rejects your entry with the message `result error` and reissues the request for ⎕-input.

Calculating the Destination of a Branch

The effect of a branch depends on the numeric value of the destination, but not at all on the manner in which the calculation is stated. So there are a great many different ways to write branch expressions. The following examples illustrate only a few possibilities.

Conditional Branch

The destination expression includes a label and a test; the result is either empty (no branch) or the value of the label. The test yields either a `1` (condition is true) or a `0` (condition is false). The result of the test selects or ignores the value of the label, through compression, reshape, take, or drop. For example, to branch to *label* when *a* is greater than *b* could be written:

```
→(a>b)/label
```

A complementary pair corresponding to *if* and *unless* can be formed with ↑ for *if* and ↓ for *unless*:

```
→(a>b)↑label      ⍝Branch to label if a>b
→(a≤b)↓label      ⍝Branch to label unless a≤b
```

Conditional Execution of Statements on the Same Line

When you want to execute *work* only when *a* is greater than *b*, you could exit conditionally from the line, thereby executing *work* only when the destination is empty:

```
→(a≤b)/next ◇ work
next: ...
```

The same effect could be achieved by

```
⍎(a≤b)/'work'
```

which may or may not be easier to read and may be less efficient to execute.

Multi-way Branch

Compression or indexing may select from among any number of alternatives. For example:

```
→((test1, test2, test3)/L1, L2, L3),default
```

or

```
→(label1, label2, label3)[test]
```

Branch with Counter

A simple loop often combines a conditional branch and an unconditional branch. The following shows a *leading decision* that controls *n* repetitions of a step called *work*. It is called a *leading* decision because the test is made before *work* is ever executed. That allows for the possibility that *work* may be executed zero times.

```
        i←0
test:   i←i+1
       →(n<i)/next
        work
       →test
next:   . . .
```

(Some people like to combine the test and the increment in a single statement, for example →(*n*<*i*←*i*+1)/*next*. However, in that form the statement is not restartable.)

Stop and Trace Controls

As an aid to debugging, you can set *stop-* or *trace-controls* for an unlocked user-defined verb. The two are independent but controlled by a similar mechanism. Stop and trace are effected by a dummy name composed of the characters *s*Δ or *t*Δ followed by the name of the verb. For example, to set stops on lines 3 and 15 of a verb called *work*, you execute:

```
sΔwork←3 15
```

Similarly, to trace the execution of all lines of the verb called *work* that has no more than 99 lines, you might execute:

```
tΔwork←⍳99
```

The stop or trace is effective for those lines that actually exist and are assigned to the stop- or trace-control (so it is not wrong to assign ⍳99 even when *work* in fact has only two lines).

Each time you set the stop-control for a verb, you thereby remove any previous stops. Similarly, each time you set the trace-control for a verb, you thereby remove any previous trace-controls. Once a stop or trace has

been set, there is no mechanism to report the numbers of the lines on which it is effective.[7]

Stops and traces are removed by setting them to an empty vector:

```
tΔwork←⍳0
```

Stop and trace have no effect on locked verbs.

Effects of Stop

When SHARP APL comes to a line marked with a stop-control, execution halts *before* work on the line is started. The halt is event `1001`. The effect is much the same as a weak interrupt signaled from the keyboard. That is, (provided the event is not trapped) execution is suspended at the beginning of the line. SHARP APL displays the name of the verb and the number of the line that is next to be executed.

You may examine the values of names, perform tests, edit definitions, and so on. When satisfied, you may resume execution at the point where work was halted by `→⎕lc` (or exploit any of the other options available when a verb's execution is suspended).

To permit the restart of suspended verbs, a line branched to from immediate execution is not subject to `sΔ`.

Effects of Trace

For each line for which a trace was requested, as SHARP APL completes execution of each sentence in the line, it displays a report. For the first sentence on each line, the report shows the name of the active verb together with the line number (in brackets). For subsequent sentences on the same line, it shows a diamond.

For each completed sentence in a traced line, SHARP APL displays the root value. For example, in a sentence such as

```
x←(a←15)×(b←22)
```

SHARP APL reports the value assigned to `x` but not the values assigned to `a` or `b`.

When the root value is an item or a list, it appears on the same line as the verb name and line number, or the diamond. When the root value is a table (or higher-rank array), the display starts on a new line (to preserve the alignment of columns).

---

7 You could pass the result of ← to another pronoun (and thereby record the values you used) by an expression such as `traced←tΔwork←⍳99`.

When a sentence has no root value (because the root verb returns no result, or the line is empty), the trace display is blank.

When a sentence explicitly calls for display, SHARP APL first does the display in the usual way, and then the trace display as described in the preceding paragraphs (so tracing a line that itself produces a display will cause two displays).

When the traced sentence is a branch, the trace display shows the right arrow followed by the numeric value of the destination.

Effect of Editing on Trace or Display

Editing a verb's definition with the ∇-editor does not affect the stop or trace-controls. If you insert or delete lines ahead of the lines that are marked for stop or trace, tracing remains with the lines originally marked, even when they are renumbered.

The verbs `□fx` and `3 □fd` in effect destroy the former definition and create a new one, so that editing a definition with either `□fx` or `3 □fd` loses all of that verb's stop or trace settings.

Effect of Storing and Retrieving a Verb with Stop or Trace Set

A verb stored in a saved workspace retains the stop or trace settings it had in the active workspace and retains them when the workspace is loaded or when the verb is copied from the saved workspace.

A verb inserted into a package by the system verb `□pack` retains its trace or stop settings and they are retained when the verb is subsequently materialized from within the package by the system verb `□pdef`.

10 *Event Handling*

An *event* is something that disrupts the normal sequence of work and requires further instructions.

An event may be *trapped* or *untrapped*. An untrapped event returns control to the keyboard for you to take appropriate action. (A task that has no interactive control – that is, an N-task or a B-task – has no way for you to tell it what to do next, and after an untrapped event simply terminates, saving its active workspace, unless instructed otherwise.)

A trapped event is one you have foreseen and provided for. The recovery instructions you store as part of a trap definition usually contain the same sorts of things you might otherwise enter from the keyboard after the event occurs. When they are stored in advance in `⎕trap`, the recovery proceeds automatically; there is no display of an error message and no pause to receive new instructions.

Errors and Interrupts

An event falls into one of two main classes:

Error SHARP APL finds that it is unable to carry out the sentence it is working on.

An error may arise for any of a number of reasons. For example, it might be because something is inherently wrong with the sentence itself (perhaps invalid syntax). The data supplied may be inappropriate to the verb (having the wrong type, rank, or value). The sentence may refer to a name that has not been defined. Alternatively, the problem may be that, although there is nothing wrong with the sentence, SHARP APL cannot execute it because of some constraint of resources (not enough room in the active workspace, file full, access denied, and so on).

Interrupt SHARP APL receives a signal from outside the active workspace indicating that work should be interrupted.

For a T-task, an interrupt most commonly arises from a signal from the keyboard: you press a key, such as *Break* or *PA1*, indicating that you want work halted, or enter an input interrupt.[1]

[1] See the description of strong and weak interrupt in Chapter 9, "Control of Execution".

For an S-task, there is a corresponding signal transmitted by way of the controlling shared noun. In addition, the system itself may generate certain signals and interrupts (for example, a warning that the system is shutting down imminently); such signals may proceed to any T-, S-, N-, or B-task.

Classifying Interrupts

An interrupt is classified by what SHARP APL was doing at the time it halted execution in response to the interrupt. For example, if SHARP APL was waiting to access a shared noun, it calls the interrupt an `sv interrupt` (for *shared variable*); if it was processing keyboard input, it calls the interrupt an `input interrupt`; if it was waiting for a return from the file system, it calls the interrupt a `file interrupt`; and so on. The term *interrupt* (without a qualifying adjective) refers to an interruption during execution of an APL expression that did not involve something external to the workspace.

Weak and Strong Interrupts

In processing an interrupt, the system recognizes two levels of urgency. The first time you send an interrupt signal, the system notes a *weak interrupt.* A weak interrupt is also called an *attention* signal. SHARP APL takes a weak interrupt to mean "Halt before starting to execute a new line."

During the interval *after* you have sent the first interrupt signal but *before* SHARP APL has halted work, you may signal a second interrupt. (It must be sufficiently delayed from the first to be distinct from it.) When the system detects a second interrupt, it upgrades the urgency of your request and calls it a *strong interrupt.* Following a strong interrupt, SHARP APL halts as quickly as possible. It is very possible that the halt resulting from a strong interrupt occurs in the middle of a line, with some portion executed and some not. This may make the verb non-restartable, in the sense that *side effects* may have occurred.

Strong Interrupts and Restartability

Strong interrupts can make a sentence non-restartable, due to side effects. How can this occur? Consider a sentence which is part of a verb to read the contents of file and catenate its contents:

```
cmp←cmp,□read tn, cn←cn+1
```

The `□read` might take a minute or more to execute if the file being read is in the process of being backed up by the system at the time the `□read` is in progress. An impatient user might signal an interrupt during the execution

of this line. A weak interrupt would wait until the line had completed execution, and suspend on the next line of the verb, which would not be a problem. However, a strong interrupt would suspend *after* the increment of `cn`, but *before* the `⎕read` had completed, resulting in suspension on *that* line, rather than on the *next* one.

The problem with suspension at that point is that the partially executed part of the line had the side effect of changing the value of `cn`, even though the remainder of the line failed to complete. This means that if the user (or a `⎕trap` expression or other method of restarting that line) branched to the line above, without correcting the value of `cn`, that the result would fail to include the contents of component `cn`.

One solution to this sort of potential problem is to isolate lines which are areas of possible significant delay, such as file operations, computationally intense expressions, and shared variable operations, from lines which which have side effects, such as multiple assignments and external operations such as other file operations or shared variable operations.

Signaled Errors

Executing the system verb `⎕signal` produces the same effect as an error. (`⎕signal` is limited to signaling errors; it cannot signal an interrupt.) In particular, `⎕signal`:

- Halts execution of the item at the top of the state indicator; that is, of the active user-defined verb or a use of ⍎ or ⎕-input. It also removes the item from the state indicator, causing the signaled error to be made visible to the verb which invoked the error-producing item. For a user-defined verb, this is equally true regardless of whether:
  - the active verb is running (that is, being executed)

    or
  - the active verb is suspended, and you enter the sentence containing `⎕signal` from the keyboard.
- Sets `⎕er` to indicate the error number specified in the right argument of `⎕signal`. The optional left argument to `⎕signal` supplies the error text in `⎕er`.

You are free to use any error number between 1 and 999. That is, your use of `⎕signal` may either emulate a standard error (such as 11, `domain error` or 3, `index error`), or it may signal some arbitrary error unknown to SHARP APL but meaningful to your application. By convention, a user-defined error is given a number between 500 and 999, whereas no system-generated error has a number in that range.

One reason for signaling a user-defined error is that you plan to trap it and thus provide automatic recovery from some condition that your application has signaled.

When `⎕signal` is used from within *execute*, or from within ⍞-input, the top line of the state indicator contains ⍎ or ⍞. Then ⍎ or ⍞ is removed from the top of the state indicator (not the user-defined verb that contains it), and the event is signaled to the next level. Thus the effect of `⍎'⎕signal n'` is not the same as the effect of `⎕signal n`.

Trap, Action, and Recovery

An event trap is a list of instructions whose effect is to say to SHARP APL, "If the following occurs, here is what to do." The trap specifies what event (or events) to look for and an action code specifying what action to take when one of them occurs.

You write these instructions by setting the value of the system noun `⎕trap`. As soon as you set a value for `⎕trap`, SHARP APL verifies that `⎕trap` contains a valid action code and well-formed event numbers. (If SHARP APL finds your `⎕trap` ill-formed or otherwise invalid, it resets `⎕trap` to be an empty list.) Apart from that initial check for an ill-formed trap, setting `⎕trap` has no effect until a trappable event occurs. Then SHARP APL looks in the various values of `⎕trap` for an event number that matches the event that has occurred.

When you set `⎕trap`, you may supply any number of separate statements about events and actions. These statements are in a special form required for `⎕trap`. The value of `⎕trap` may be either a table, with one row for each trap statement, or a list. When it is a list, the first character in the list serves to delimit the various trap statements. (See the description of `⎕trap` in Chapter 16, "System Nouns and Verbs".)

Fields within Each Trap Statement

Within each trap statement, there are four fields. The fields require no delimiter, but it is common to put blanks between them to make them easier to read. Only two of the fields are required: *event* and *action*. The four fields are as follows:

Events A number or list of numbers to which the trap applies. (Details of the way event numbers may be written are included in the description of `⎕trap` in Chapter 16, "System Nouns and Verbs".)

Own The optional letter `o` (for *own*) restricts the trap to events that occur at the level of the verb to which `⎕trap` is local. That is,

the presence of an `o` excludes events encountered by sub-verbs (user-defined verbs invoked by the verb to which `□trap` is local).

Action A single letter identifying one of the classes of action to be taken when the event occurs. The actions are described later in this chapter. The letters and their meanings are:

`s` ***Stop.*** Causes the event to treated in the interpreter's default manner. The event numbers may be omitted, in which case *stop* applies to *any* event.

`n` ***Next.*** Look no further in this `□trap` (but continue the search at more global levels, if any). This action makes it easier to construct an exception to a trap, in the form, "For event x, look no further; for the remaining set of events (which otherwise would include x), continue to look further."

`e` ***Execute.*** Execute the recovery expression.

`c` ***Cut.*** Cut back the state indicator by halting any verbs invoked by the verb that localized the relevant `□trap`, and execute the recovery expression, at the level at which the localization occurred.

`i` ***Immediate.*** Allows you to supply the value of the missing name (by executing the recovery expression) and resume execution of the sentence at the point at which it was interrupted.

Note that action `i` is in several ways unlike all other actions:

- It is usable with only one event, event 6, `value error`.
- Resumption occurs in *mid-line*, with the missing name.
- There is never even a nominal suspension (so the top line of the state indicator never has an asterisk).
- Resumption is automatic, *without* a branch in the recovery expression.

`d` ***Do.*** Carry out one of several actions identified in the keyword that follows the letter `d`. The actions permit steps that cannot be stated as APL expressions.

Recovery For actions `e`, `c`, or `i`, this last field contains a line of APL called the *recovery expression*. The line may contain diamonds, and may invoke other verbs. If action is to be restarted, the recovery expression for actions `e` or `c` must include a branch statement. Action `i` restarts automatically in mid-line and does *not* use a branch statement. For actions `n` and `s`, this

field is empty. For `d`, this field contains one of the words `clear`, `exit`, or `off`, indicating an action to be performed. (See the discussion of `⎕trap` in Chapter 16, "System Nouns and Verbs".)

Unreportable or Untrappable Events

Certain events are outside the scope of the event handling mechanism. In particular, nothing that occurs during execution of a system command can generate a reportable or trappable event. Certain actions of the operating system – for example, termination of the APL task by system shutdown or `⎕bounce` – are, in principle, trappable, but there may be little you can do to recover from them.[2]

An error *directly* encountered during execution of a recovery expression is not itself trappable. This restriction prevents endless recursion from certain defective `⎕trap` expressions. However, even with this feature, it is still possible to construct event handlers that execute endlessly, so prudence is called for in their creation. Furthermore, errors occurring during the execution of a user-defined verb invoked by a `⎕trap` expression *are* trappable.

Event 2001 (return to immediate execution) is not reportable – that is, does *not* set a new value for `⎕er`, even though it *is* a trappable event.

Events 67 and 68 (certain system errors) are reportable – that is, *do* cause `⎕er` to be set, even though it is not feasible to do anything about them. They are not trappable, except when generated via `⎕signal`.

An event arising during execution of a system command is never reported in `⎕er` and cannot be trapped.

Environment Condition Prohibits Suspension

There arise cases in which you do not want a verb to be suspended. For example, in a shared nouns application, you do not want to permit one user, by interrupting a verb, to tie up another user. To preserve the security of data in a shared file, you do not want a user to be able to examine nouns before the verb has completed its processing. One way to prevent a verb from being suspended is to trap the events that might suspend it. However, such a trap can take effect only when it has been set and only when its recovery expression deals successfully with the event. Suppose the event occurs in the interval after the verb has started execution but before a trap has been set. Suppose that when the event occurs, it is something for which

[2] If you were to trap a "bounce" action, about the only feasible action is `d clear`, which would prevent saving of a `continue` workspace. It might also be used to perform a little tidying up before termination.

the trap makes no provision. □*ec* (for *environment condition*) provides a more certain means to prevent suspension.

□*ec* is a *system shared noun*. That is, it is always shared with the SHARP APL system, and SHARP APL can set it. In particular, SHARP APL always sets □*ec* so that it is never without a value. Within any verb to which □*ec* is local, *SHARP APL immediately sets its value to* `1`. The verb may then contain a statement such as `□ec←0`. However, it is not necessary to set a initial value `□ec←1` because *merely by making* □*ec local, you direct SHARP APL to set it to* `1`.

Within a verb whose definition localizes □*ec*, there is *no* interval during which an event could occur before the verb has had time to set □*ec* to `1`, since it automatically has that value by virtue of being localized.

Effect of `1` as the Value of □*ec*

As long as the value of □*ec* is `1` at any level of the state indicator, a user-defined verb cannot be suspended *unless* the current line is one for which the *s*Δ stop-control has previously been set. Thus, unless stop-control has been set, an event must either:

- be successfully trapped

  or

- cause the active verb to be abandoned, and the event signaled to the environment that invoked it.

Consider the behavior of a verb *foo*, whose definition is as follows:

```
    ∇ z←α foo ω
[1]   z←ω÷α
    ∇
```

When α is `0` but ω is not, the arguments are outside the domain of ÷. That is a domain error. How SHARP APL responds to that event depends on the value of □*ec*, as follows:

```
      □ec←0
      0 foo 12
domain error
foo[1] z←ω÷a
           ∧
)si
foo[1]
```

```
      □ec←1
      0 foo 12
domain error
      0 foo 12
        ∧
)si
```

When □*ec* is `0` (its default value in a clear workspace), the verb *foo* halts on line 1, and the error report points to the symbol ÷. But when □*ec* is `1`,

execution of *foo* is halted. The event is still reported as *domain error*, but is attributed to *foo* rather than to a sentence within it.

To set □*ec* globally to 1 is usually more protection than is needed. Instead, you may set it to 1 only for some critical section of a definition. That can be achieved by a sequence such as the following:

```
[ ]   [non-critical work...]
[ ]   □ec←1
[ ]   critical section
[ ]   □ec←0
[ ]   [more non-critical work...]
```

Values of □*ec*

□*ec* is binary: you may set it only to 1 or to 0. In a clear workspace, the default value of □*ec* is 0. If you set it to anything other than an item 1 or 0, SHARP APL resets it to 0.

□*ec*←1 in the SI Stack Prohibits Suspension

Like □*trap*, □*ec* has the important property that SHARP APL does not consider only the *visible* (that is, most local) value, but *any* value, at *any* level of the state indicator. As long as *any* □*ec* has the value 1, the *only* untrapped event that can cause execution to be suspended is a stop-control, set prior to the event by *s*Δ.

To visualize the effects of □*ec* and □*trap* at various levels, consider the diagram in Figure 10–1. It assumes that *verb*1 invokes *verb*2 which invokes *verb*3 which invokes *verb*4. Since the state indicator shows the most recently invoked verb at the top, *verb*4 is shown at the top and *verb*1 at the bottom. An additional row at the bottom shows the global environment before *verb*1 was invoked.

The column at the center indicates where □*ec* has been localized and shows the values to which it has been set. The column at the right shows the effective value of □*ec*. The behavior of □*ec* is unlike that of any other name. While the value of □*ec* is 1 at *any* level of the state indicator (regardless of whether it is explicitly set to 1 or has the value 1 by default), its effective value at all higher levels of the state indicator is 1. Thus the effective value may be controlled by a □*ec* that the higher levels can neither see nor set. (Compare Figure 10–1 with Figure 9–1; the 1 in the □*ec* localized by *verb*2 is effective despite being masked by those localized in *verb*3 and *verb*4.)

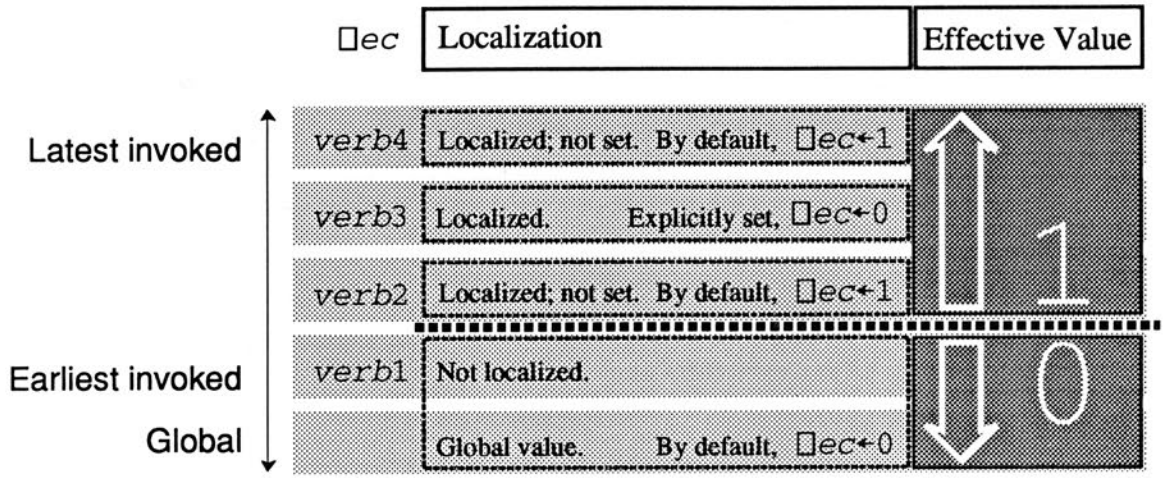


Figure 10–1 : Effective value of □ec *is the maximum* □ec.

Since □ec is localized by verb2, no event can cause suspension in it or in verb3 or verb4, regardless of how verb3 or verb4 may set their own local values of □ec.

Response to an Event When Stop is Prohibited

What happens when a verb that is not allowed to stop does not know how to continue?

During execution of a verb *v* in an environment where the effective value of □ec is 1, an event has the following effect:

Stop Control (event 1001)	Halts execution in the usual way.
Weak Interrupt (event 1002)	SHARP APL notes the interrupt but postpones acting on it until the effective □ec becomes 0 (which might not be until it completes execution of the verb that localized the effective □ec). As a consequence, a weak interrupt has its effect on the verb that invoked *v* rather than on *v* itself. Although delayed, the weak interrupt has been noted; a second interrupt signal can still upgrade it to a strong interrupt.
Error or Strong Interrupt (except events 1001 or 1002)	Execution of *v* is halted, and the error that occurred in *v* is instead signaled to the verb that invoked *v*.

What Happens When an Event Occurs

When SHARP APL detects an event (that is, an error or an interrupt), it does the following:

Notes the event for `⎕er` Eventually, SHARP APL records the event in `⎕er`. However, this record goes into the `⎕er` visible at the level at which the event is handled. If the event is re-signaled to the next level (because stopping is not permitted at the present level), the note of the event is carried with it, as many times as necessary, until the level at which there is a suspension or the event is finally trapped.

Checks `⎕trap` SHARP APL searches the visible (that is, most local) value of `⎕trap` for a trap that covers the event number just encountered. It searches `⎕trap` from top to bottom, left to right. The first trap which covers the event is used.

When the `⎕trap` being searched contains a trap statement that applies to this event, SHARP APL carries out the indicated recovery action. If that is sufficient to resume execution, execution resumes and there is no further consideration of `⎕trap` or `⎕ec`.

Checks `⎕ec` When no applicable trap is visible, or there is one but its execution does not cause a resumption of execution, SHARP APL checks the value of `⎕ec` at every level of the state indicator from the current level down. If `⎕ec` is *not* `1` at *any* level, execution halts.

If `⎕ec` anywhere is `1`, SHARP APL takes the exit procedure described below.

Checks further When no relevant trap has been found but `⎕ec` is everywhere `0`, SHARP APL digs deeper in its search for an applicable `⎕trap`. It examines the next level at which `⎕trap` has been localized.

If at the next level it finds an applicable trap that resumes execution, it takes that action. But if it does not find one, it repeats the cycle, considering the next level at which `⎕trap` has been localized, until it reaches the global `⎕trap` value.

Exit When at *any* of the levels checked:

- the event has *not* been handled by a trap

  and

- `⎕ec` anywhere in the state indicator has the value `1`,

SHARP APL abandons execution of the verb at the top of the state indicator and signals the same event to the verb that invoked it (that is, the verb next below it on the state indicator).

This new event starts the examination of trap and environment all over again, possibly cascading further down the state indicator, and possibly reaching a level at which `□ec` is no longer `1`.

Halt Only when it has checked back through all levels of the state indicator, and at each level has found no applicable trap that resumes execution, *and* has worked down to a level at which `□ec` is no longer `1`, does SHARP APL suspend work with a message describing the event.

Then SHARP APL sets `□er` with the event number passed back through the various levels. The event number is that of the original event. However, the event is attributed to the verb in which the event was most recently signaled. When there was a cascade of events because the original event took place at a level where stopping was prohibited, that information is discarded; only the most recent level shows.

Once execution stops, you may edit the definition of the pending or suspended verbs (see Chapter 8, "Verb Formation"), restart or halt the suspended verb, or enter a new sentence for execution (see Chapter 9, "Control of Execution").

Caution: Trapping Interrupts or Prohibiting Suspension

Prohibiting suspensions can make debugging difficult, since it becomes impossible to see where the precipitating event took place. However, it is relatively simple to use no `□ec` in the main body of definitions, and set `□ec` to `1` only in an outer verb which invokes the others.

Since `□ec` is in that case global to the verbs that do the work, if you do not wish users of your application to be able to suspend it, you must prevent them from setting stop controls on your verbs.[3] You can do that by locking the definitions, or by making their names local to the calling function and reading them from a file during execution (so that they have no global existence in the workspace).

Trapping interrupts is sensitive. Interruption is the means by which a program's user or designer reasserts control when things are going wrong. You

[3] Editing the definitions (for example, to insert `□ec←0`) would make a difference only if your application had no more than a single level of `□ec` localization, so that the visible `□ec` was the only `□ec`.

may trap interrupts in order to achieve a more graceful recovery, but be aware that a program in which interrupts have been trapped improperly may be impossible to stop. You may have to disconnect the terminal, or sign on another terminal and use `⎕bounce`, or request the operator to cancel the task. In the time required to take those steps, a runaway task can do a great deal of computing.

Caution: After Suspension, `⎕trap` Is Still Active

When you have control at the keyboard, but there is a suspended verb on the state indicator, even though SHARP APL did *not* find a trap for the event that caused execution to suspend, `⎕trap` is still there *and may respond to errors in what you enter now*. If you enter a new error and the new error *is* trapped, the recovery expression may abandon the suspended verb or resume the suspended verb's execution.

When you want to experiment following an error, it is prudent to save the visible `⎕trap` and reset it temporarily to something that causes all events to behave as if no `⎕trap`s were present (for example, `⎕trap←'⍬ s'`).

11 *Shared Variables*

This chapter discusses shared variables, the Shared Variable Processor, and how shared variables permit one APL task to communicate with another APL task or with processes outside of APL.

Shared variables, or shared names, are the means of communication between the active workspace of an APL task and other APL tasks or *auxiliary processors* that are provided with the SHARP APL system.

A Discipline of Communication

The sharing of names was introduced by IBM in APL/SV under the title *shared variables*. A *shared name* identifies a *noun* whose value can be set and used in *two different contexts simultaneously*. (In APL terms, such a context is a workspace; but a name can be shared with a program outside APL, where the term "workspace" may not apply.) Two otherwise independent programs, one in each of those two contexts, can interact; when one of them sets a new value for the name they share, that new value is visible to the other.

For example, suppose Task 1 and Task 2 represent the active workspaces of two APL users who have agreed to share the name *x*. The left side of Figure 11–1 shows events in Task 1's workspace, and the right side events in Task 2's workspace. The fact that *x* is shared leads to the following simple example (which would otherwise be quite surprising). Task 2 assigns to *x* the value `abc def`. A moment later, Task 2 evaluates `2×x` and gets the answer `10`.

Time	*Active workspace of task 1*	*Active workspace of task 2*
↓		`x←'abc def'`
	`x←5`	
		`2×x`
		`10`

Figure 11–1 : Shared variable interactions between tasks.

Of course the reason for this result is that after Task 2 gave *x* the value `abc def`, Task 1 gave *x* a new value. Since *x* is shared, Task 2 now sees the *new* value.

There is a simple but rigorous set of rules governing communication through shared names. This chapter introduces those rules and the system nouns and verbs that embody them.

Two processes that are running independently but concurrently can be linked by sharing a name between them (or sometimes two names, or many names, depending on what is to be done). Each of the communicating tasks starts by *offering to share* a name[1] with the other. When each offers the other the *same* name, they are said to have made *matching* offers. Once they have made matching offers, the matching name is *shared* between them. (It does not matter who offered the name first and who second.)

Once sharing has been established, when one of them gives a new value to the name they share (that is, *sets* it), the other has only to *use* that name to become aware of its new value. For example, from an APL workspace, a transaction might look like this:

```
x←'command'          ⍝ Set the name.
result←x             ⍝ Use the name.
```

When *x* is shared, using *x* in the second statement may reveal not the command written there in the first statement, but a new value supplied by the processor with which *x* is shared. Typically, you set *x* with the text of a command, and you use *x* to receive an acknowledgment or result.

The Shared Variable Processor

The Shared Variable Processor (SVP) makes it possible for the same name to appear in two otherwise separate contexts. The SVP manages all use of shared names for any of the APL users, and also for non-APL jobs that may use shared names. It maintains tables of the names that are shared and the tasks that are sharing them. It provides intermediate storage for their values while one partner has set a value that the other has not yet used. It enforces the rules that prevent conflict over whose turn it is to set the name or to use it.

Sharing with Non-APL Programs

Sharing can be established between *any* pair of tasks that abide by the shared name discipline. The active workspace of an APL task can share

[1] Or a set of names. For simplicity, the discussion talks about sharing one name, but the offer may be for a set of names.

with the active workspace of any other APL task, and with almost any sort of program, even one written in a completely different language. In that manner, your APL workspace can send commands to the operating system to execute certain MVS commands or to make use of non-APL files.[2]

Auxiliary Processors

An *auxiliary processor* (AP) is a program that provides a service to an APL task but which may not itself be part of APL; it uses shared names to communicate with the APL task. Each AP runs at the same time as SHARP APL and communicates with SHARP APL through the Shared Variable Processor. Thus APs give SHARP APL access to external facilities such as databases and language processors.

How Name-Sharing Works

The partners that share a name may be a pair of APL tasks, a pair of APs, or an APL task and an AP. Each of the partners that share a name keeps its own copy of the name's value, but marks the name as "shared." Whenever either of them refers to the shared name, it checks first with the Shared Variable Processor. The SVP notes the new value each time that either partner sets it. When one of the partners makes use of the shared name, the SVP supplies its current value.

This will become clearer with an example. Suppose you have arranged to share the name *a* in your APL workspace with a partner that will carry out commands for you. (The partner might be another APL task, or it might be an auxiliary processor; the process is the same in either case.) You assign a value to *a* so that your partner will see it and treat it as a command to be executed, as follows:

- You establish that *a* is shared between your APL workspace and your partner. (The details of this step are described in the section "Establishing a Shared Name", below.) Inside your workspace, the interpreter marks the name *a* to show that it refers to a shared name.
- You assign a value to *a*. When the APL interpreter executes that assignment, it notices that *a* is a shared name and notifies the SVP that a new value has been received. The SVP stores its own record of the value of *a*. (For the moment, the value exists both in your workspace and in the SVP's own storage.) The SVP notes that you know the latest value of shared name *a* (since you put it there) but that your partner does not yet know the value.

[2] Auxiliary processors may also be used to communicate with specialized processors to perform a variety of tasks (for example, to perform specialized calculations, to manage external communications, or to provide special services for managing the screen).

- Frequently (but not necessarily) a partner task does not do work of its own, but waits for orders. It "sleeps" until notified by the SVP that something has happened to affect a name it is sharing. Then (perhaps after some initial checking to see what has happened) it makes use of the variable it shares with you. When it does, the SVP transfers to it the duplicate value it has been holding. The SVP notes that both sharers are now aware of the value of *a*, and discards its duplicate. Now the shared name exists with the same value in your APL workspace and your partner's.
- In due course, your partner has a reply for you. Your partner sets the value of the shared name. The SVP notes that your partner has done so, and saves a record of the shared name's new value. The SVP notes that your partner now knows what the value is (having just set it) but you do not (because you have not used the name yet).
- You use your name *a* again. (You may have been waiting for the reply, or you may have gone on to do something else, not coming back to look at *a* until later.) When you next use *a*, the SVP supplies the value it has been holding, and notes that both partners are now aware of its value.

That is how the Shared Variable Processor makes *a* seem to exist in both workspaces. The value is not in fact transferred until you attempt to use it, but that is not apparent because any time you ask, "What is the value of *a*?" the SVP slips in and delivers the value it has been holding.

Sending and Receiving

When two tasks communicate by sharing a name, there is no special action called *send*. Nor is there one called *receive*. To send some data to a partner, you use the arrow ← to assign a value to the name that you have shared with the partner. That name is visible to the partner too, and so you have "sent" the value just by giving it a value in your own workspace.

Similarly, to receive data from your partner, you have only to use the value of the shared name that you find in your own workspace: that is where the SVP delivers it to you.

Set, Use, and Reference

A name is *set* when it occurs immediately to the left of ←, the copula. Any other occurrence is a *use* of the name. As will be apparent later, it is also useful to have a term that includes both actions: *reference* means any occurrence of either set or use.

Establishing a Shared Name

These are the steps that each program that uses shared names must go through. Your application may embed them in a cover verb so that the steps are no longer apparent each time. (The corresponding steps are just the same, regardless of whether the task with which you share is another APL task or an AP.)

- Using the verb `□svn` (shared variable name), you set the identifier which distinguishes multiple tasks for the same user number.
- Using the verb `□svo` (shared variable offer), you offer to share a name with another task. The SVP notes your offer. If you are the first of the partners to offer the name, the SVP gives you a result of `1`, meaning that it has received one offer to share this name (which so far is unreciprocated.)
- The SVP makes your offer available to your proposed partner. When the partner inquires, "Has anyone offered to share with me?" the Shared Variable Processor informs it that your task has made an offer.

  When your proposed partner is an APL task that you started, it is probably written so that it is expecting your offer. Moreover, it is not interested in offers from anyone else, and so need not check for other offers.

  When your proposed partner is an AP, or a public or semi-public server task, it probably does not have any special expectations about who will offer to share with it or when such offers may arrive. But usually it has nothing else to do but serve tasks that offer to share with it. Such a task is usually written so that it is permanently vigilant for new offers to share, and hence notices promptly when you extend an offer to it.
- The prospective partner responds with a matching counter offer. That is, it offers to you the same name that you offered to it. You can tell that it has done so because when you now use `□svo` with respect to the name you offered, the result is `2`, when before it was `1`. When both sides have made matching offers to share the same name, full sharing is in effect: the *degree of coupling between the two processors is* `2`.
- Using the verb `□svc`, you or your partner set up the *shared variable control* for the name you have shared. This interlocks references to the shared name so that you and your partner do not get out of step; it prevents misunderstandings that would arise if one partner were to rush ahead instead of waiting for the other.

- Now begins a cycle that may be repeated many times as you work with your partner task:
  - You set the shared name. For example, you set it with the text of a command you want the partner to process.
  - Your partner uses the value you have set, and does what is called for.
  - Your partner sets the shared name, either with an acknowledgment of your request or with an answer to it.
  - You *use* the shared name to find out what your partner's response was.
- At any point in the process, you can use the verb `⎕svs` to inquire about the *shared variable state* of the name you have shared. This gives you the answer to the question, "Who does not yet know the shared name's latest value?" Thus, (if the need arises) you can tell whether your partner has used what you set, or has set a reply for you to use.
- If you are waiting for a reply from several different shared names, but you do not know which of them will reply first, you can have your program wait for a *state change* by using the system noun `⎕sc`. When you refer to `⎕sc`, your program is suspended until there is a change in the *state* of one or another of the names you have shared, until a `⎕sc` *time-out* occurs, or until some task makes a new offer to you, or until a system shutdown warning post occurs.

Offers and Retractions

The dyad `⎕svo` offers to share names with a processor. The right argument may be a single name or a table of names or pairs of names. The left argument indicates the processes with which sharing is desired. (For details of the syntax, the form of the arguments, and so on, see the entry under each verb's name in Chapter 16, "System Nouns and Verbs".)

Surrogate Names

You may find it inconvenient to assign the same name as your partner. For example, a server task might expect to share the name `command`, but you have something else called `command`. In such a situation, you can make your offer with a *surrogate name*, which looks like this:

```
Processor ⎕svo 'command wish'
```

In effect, you are saying, "I offer to share a name that as far as I am concerned is `command`, but that you may think of as `wish`."

The proposed partner may do something similar. Perhaps the partner's program really thinks of this name as `input`, but offers to let you call

it `wish`. These offers match, and sharing goes into effect. Your name `command` is linked to the partner's name `input`. Each of you may think the other is sharing `wish`; neither of you need know that the other privately calls the name something else.

Processor ID, Clone ID, and System ID

The partner to whom you are making an offer is identified by a list of numbers, called the partner's *processor ID*. If two numbers are specified, the first number identifies the "owner" of the processor and the second is a *clone ID*, for use when the same owner has several tasks running at once. If three numbers are specified, the first is a *system ID* specifying the system to which sharing is offered.

When the partner is an APL task, the number that represents the owner is its APL *user number*. When the partner is an AP, the "owner" is an arbitrary number assigned by the system administrators (such as the number 1 for the APL S-task processor, or 370 for the TSIO AP). It is customary to use numbers greater than 1000 to represent individuals, and lower numbers to represent auxiliary processors (APs).

The second number of the pair, the clone ID, is used to distinguish between different tasks that have the same owner. For example, when you have two concurrent APL tasks, they have the same owner. If they both use shared names, they have to distinguish themselves.

A task does not have to set a clone ID unless one is actually needed. Each task sets its own clone ID. If you use shared names but do not state a clone ID, it is assumed to be zero. That may be adequate provided you or whoever owns the task you want to share with has only one task using shared names. However, a program that regularly uses shared names should not rely on the default, but should establish its own clone ID. Since every task also has a unique task ID, it is common practice for a task to adopt its task ID as its clone ID.[3]

Degree of Coupling

The number of partners sharing a name is its *degree of coupling*. The degree of coupling may be `0`, `1` or `2`, meaning, respectively, "not offered," "unrequited offer," and "shared."

A name's degree of coupling is never more than `2`. Communication among more than two partners is always built from channels having two ends each.

[3] In Chapter 16, "System Nouns and Verbs", see the discussion of `⎕svn`, which sets or reports your task's clone ID, and `⎕runs`, which reports (among other things) the task ID of each of the tasks on your user number.

Duration of Sharing

Sharing goes into effect when the level of coupling becomes `2`. That is, when you and your partner have made matching offers to share a name. Sharing remains in effect until either side *retracts* the offer, either implicitly or explicitly. The system verb `⎕svr` *explicitly* retracts your offer to share a name (see its description in Chapter 16, "System Nouns and Verbs").

You *implicitly* retract an offer to share whenever the name that you offered ceases to exist. A name ceases to exist when you expunge it, or when you exit from a verb to which it is local. The name that you offered (along with all other names and everything else in your active workspace) ceases to exist when you execute the command `)clear` or `)load`. And of course, it definitely ceases to exist when you terminate the task (for example, by `)off`).

The offer to share a name is also implicitly retracted when you create the name anew by using `)copy` to import it from a saved workspace, or by using `⎕pdef` to define it from a package.

It is standard practice to write an AP so that, when it finds that you have retracted your offer to share, it assumes a normal end to its transactions. It withdraws its outstanding offer, and tidies up whatever it was doing. For example, if you have shared a name with the TSIO AP and used it to open a file, and then you erase your shared name (or do anything else that terminates the share), TSIO closes the file for you.

Localization and Shared Names

It is permissible to share a local name, but when you do, the sharing ceases as soon as you exit from the verb to which the name is local. So you probably do not want to do that unless the verb to which your shared name is local is going to become active rarely enough, or remain active long enough, to make it worthwhile to reinitiate sharing each time the verb is invoked.

When you have shared a global name `a`, and then, while that share is in effect, you invoke a user-defined verb that uses `a` as a local name, the local name is the only one visible. Temporarily, your global `a` is *shadowed* by the local meaning. If you execute `⎕svo a`, the result is `0`.

Shadowing a shared name does not constitute an implicit retraction. As soon as you exit from the verb to which `a` is local, you regain access to the `a` that you shared, and the result of `⎕svo a` is again `2`. IN fact, while localized, the name may itself have been reshared.

Inquiring About Offers Made to You by Others

The verb `□svq` lets you inquire about offers that others have made to you. That means offers that *they* initiated. It does not include offers they made to match outstanding offers of yours. Nor does it include general offers they may have made (using `0 □svo`).

You need information about offers made to you when you write a program that expects to collaborate with one or more tasks. (Some APs are written so they do not initiate offers, but only respond to offers that you make to them. An AP written in that way will not show up in the result of `□svq`.)

The inquiry using `□svq` has two forms. One asks, "Which processors have made offers to me?" It returns their processor IDs. The other asks, for a particular processor, "What names has that processor offered to share with me?" For details of `□svq`, see its description in Chapter 16, "System Nouns and Verbs".

The Shared Name's State and Controls

Effective communication requires:

- keeping track of who knows the shared name's current value.
- regulating whose turn it is to send or receive.

Information on who knows the shared name's current value is called the *shared variable state*. The verb `□svs` reports the state of a set of names.

Control over which partner can set or use the name they share is provided by *shared variable control*. The verb `□svc` reports or sets the control on a set of shared names.

State and control are described in a similar way. Each uses a four-item Boolean list, described in what follows.

Point of View in `□svs` and `□svc`

When two processors share a name, sharing is symmetric. Neither processor takes precedence over the other. It does not matter which made its offer first and which second. To preserve the symmetry of sharing between partners, in the shared name's state and control, each processor sees itself as the *first* member of the pair, and its partner as the second.

The four numbers that describe the state or control are treated as two pairs. The first pair concern *setting* the shared name. The second pair concern *using* it (that is, reading it to make use of its value). The significance of the four numbers can be summarized thus:

Set the Name		Use the Name	
You	Partner	You	Partner

First Pair: Who Has Set a New Value?

A `1` indicates that the task in question has set the most recent value but the other task has not yet read it. The possible values for the first pair are therefore:

`1 0` You set the value yourself (so you should know what it is), but your partner has not looked at it.

`0 1` Your partner has set the value, but you have not yet looked at it.

`0 0` You and your partner are both aware of the shared name's value, and neither has since set a new value. This is also the value when sharing has just been established, and neither partner has yet set the shared name.

Clearly, you cannot *both* have set the most recent value, so `1 1` is not possible.

Second Pair: Who Is Aware of the Current Value?

A `1` indicates the that task knows the current value. Once information is being exchanged, there are three possibilities:

`1 0` Only you know.

`0 1` Only your partner knows.

`1 1` You both know. (This is also the value when sharing is first established.)

Possible Values of □SVS

Shared names exist in four possible states, summarized by the state settings shown in Figure 11–2. Once sharing has started, there are three distinct states which correspond to the three circles in Figure 11–3. The fourth possible state, `0 0 0 0`, is reported only when the name is not shared.

Who has set a value that the other has not used?		Who knows the current value?		Interpretation
You	Partner	You	Partner	
0	0	1	1	You and your partner are both aware of the current value, and neither has since set the name. (This is the initial value when sharing is first established).
1	0	1	0	You have set the shared name, but your partner has not yet used it.
0	1	0	1	Your partner has set the shared name, but you have not yet used it.
0	0	0	0	The argument is not a shared name.

Figure 11–2 : Possible values of `⎕svs`.

Access Control

If you set the shared name with a message for your partner at the same time that your partner is doing the same for you, one of those messages is going to be lost. If you impatiently read the value of your shared name before your partner has had time to send a reply, you will be re-reading your own earlier request, or your partner's previous reply. You want to avoid that kind of confusion. Either partner may assure effective communication by setting the *access control* appropriately.

Access control works by *inhibiting* you from doing something too soon. For example, it can inhibit you from reading the reply until your partner has set it for you. Or it can inhibit you from setting a new value until your partner has read the one you set previously.

When the access control inhibits an action, SHARP APL does not reject the action, but *delays* it until the necessary conditions are met. For instance, suppose the control inhibits you from receiving the value of `sv` until your partner has set it. When you receive your partner's reply, you wish to apply

to it the verb `analyze`. Then your program should contain a sequence such as this:

```
[...]   data←sv
[...]   analyze data
```

The Shared Variable Processor delays supplying the value of `sv` until your partner has set it. You do not get an error message; referring to a name that is inhibited is not an error. However, if your partner is slow about setting `sv`, you experience a delay.

If you do not want to do anything else until your partner sends a reply, you can set the shared name and immediately refer to it. Programs of that type routinely contain a sequence such as the following:

```
[...]   sv←command
[...]   reply←sv
```

When the control has been set properly, this gives you the reply as soon as it is available. Once you have set `sv`, your program waits until the reply is ready. There is no risk of using the result too soon, nor of delaying longer than necessary.

Notice that in the two preceding examples, the value of the shared name is immediately assigned to another name. The other name is not shared. This is important. If you omit this step, you have no way to refer again to the value of your partner's reply, since using the shared name again would get the *next* value that the partner sets.[4]

You may prefer to do something else while your partner works on the command you sent. In that case the program may look more like this:

```
[ ]   sv←command
[ ]  test:→(0 1 0 1≡⎕svs'sv')/ok ⍝ Did partner reply?
[ ]   Do something else for a while
[ ]   →test
[ ]  ok:reply←sv
```

If your partner does not respond and you get impatient with the delay, you can signal a *strong interrupt* from the keyboard. That aborts your current effort to set or use a shared name. The event is 1005, `sv interrupt`. Unless the event is trapped, that returns you to immediate execution. You are free to do something else before returning to the delayed instruction, or abandon the attempt altogether.

Setting the Shared Variable Control

Access control uses a four-item list in the same format as shared variable state. By making the appropriate elements `1`, either partner may specify

[4] See also the discussion headed "Watch out" in the section "Attempting to Carry Out an Action that is Inhibited", below.

what actions are inhibited. (Before either partner has set it, the access control is `0 0 0 0`; you cannot set a name's access control until you have offered to share it.)

The four elements of the access control have significance as follows:

Inhibit re-set		Inhibit re-use	
You	Partner	You	Partner

Whenever a name's access control contains a `1`, the corresponding action is inhibited. The effect of a `1` in each of the four positions is as follows:

`[1]` You are inhibited from setting the shared name until your partner references it (and thereby knows the current value).[5]

`[2]` Your partner is inhibited from setting the shared name until you reference it (and thereby know what the current value is).[6]

`[3]` You are inhibited from using the value of the shared name until your partner has set it.

`[4]` Your partner is inhibited from using the value of the shared name until you have set it.

Effective Access Control

Either partner can inhibit an access by setting the appropriate element to `1`. The *effective* access control is the control actually in effect, as a consequence of what the partners have requested separately. The effective access control is obtained by *or*ing the access controls that the partners propose separately. (The Shared Variable Processor keeps track of which partner requested what.) The effective access control has a `1` wherever *either* partner proposed a `1`. That is, either partner is free to inhibit any of the four possible actions. However, only the partner who proposed an inhibition can turn it off.

Common Values of the Effective Access Control

`1 1 0 0` ***Waiting for read.*** Neither of you can set a new value until the other has referred to it. (Sometimes called *half duplex*.)

`0 0 1 1` ***Waiting for write.*** Neither of you can re-read the shared name until the other has set it. (Sometimes called half duplex.)

---

5 Ordinarily, your partner will discover the current value by *using* the shared name and thereby seeing the value you put there. However, your partner could reference the name by ignoring what you put there and *setting* a new value instead. Your partner will then know the current value – even if it is not the one you intended.

6 Ordinarily, you discover the current value by *using* the shared name and thereby seeing the value your partner put there. However, you could reference the name by ignoring what your partner put there and *setting* a new value instead. You will know the current value – even if it is not the one your partner intended.

`1 0 0 1` ***Waiting when you send to partner.*** You cannot re-set until your partner has referred to it, and your partner cannot re-use it until you have set. (Sometimes called *simplex* transmission from you to your partner; appropriate if you are having a rather one-way sort of conversation, as you might while sending instructions to a printer or plotter.)

`0 1 1 0` ***Waiting when partner sends to you.*** You cannot re-read until your partner has set, and your partner cannot set until you have referred to it. (Sometimes called simplex transmission from your partner to you.)

`1 1 1 1` ***Full interlock.*** Two-way conversation; each must receive and then reply. Sometimes called *reversing half duplex.* This may be the effective setting as a result of each partner setting `1 0 1 1`.

`0 1 1 1` ***You can send without waiting for your partner to read what you sent previously.*** This is usually a temporary setting. If each partner initially sets `1 0 1 1`, the effective setting is `1 1 1 1`. However, when you find that you wish to send your partner a new message without waiting for the partner to read the last one, you can remove the inhibition by `0 ⎕svc` (and restoring it afterwards).

`0 0 0 0` ***No interlock.*** Such a setting is reasonable only when you know that other constraints will prevent the two processors from getting out of step; for example, when the programs that use this shared name without interlocks also use another name that *is* interlocked.

Some servers use two shared names, one to pass *control* information, and the other to pass *data.* The control name is fully interlocked, but the data name is not interlocked at all. In such a situation, you receive data *from* the partner by using the data name *after* the partner has set the control name, but send data *to* the partner by setting the control name *after* you have set the uninterlocked data name. For example, assuming that the control name is `ctl` and the uninterlocked data name is `dat`:

```
[ ]   ctl←command ⍝ Request data from partner.
[ ]   ack←ctl ⍝ Receive partner's acknowledgment.
[ ]   test ack ⍝ Verify that partner said OK.
[ ]   reply←dat ⍝ Receive the data.

[ ]   dat←x ⍝ Prepare data intended for partner.
[ ]   ctl←command ⍝ Instruct partner to use dat.
[ ]   ack←ctl ⍝ Receive partner's acknowledgment.
[ ]   test ack ⍝ Verify that partner said OK.
```

Verifying that Access Control Has Been Set

Partners involved in two-way communication usually set access control to `1 0 1 1`. (When your partner sets `1 0 1 1`, from your point of view, that is `0 1 1 1`.) When both partners have done that, the effective access control is `1 1 1 1`. However, each partner controls his own "I set" bit, and hence can if need be turn it off (for example, to interrupt the other partner).

It is unwise to rely on your partner to set access control (even though partners such as APs usually do set access control immediately after accepting an offer to share). When you have just offered to share a name with a server task or an AP, you do not want your first instruction to be lost. Since you cannot be certain when the AP will respond and will set its side of the access control, you should not assume that access control has been taken care of; the AP may not yet have had time to do so when you are ready to send your first command. It is prudent *always* to start by setting the access control yourself.

Graphic Representation of Communication Control

The three circles in Figure 11–3 represent the three states of the shared name; the caption describes each state in English. Inside each circle you see the four-element list representing the name's state.

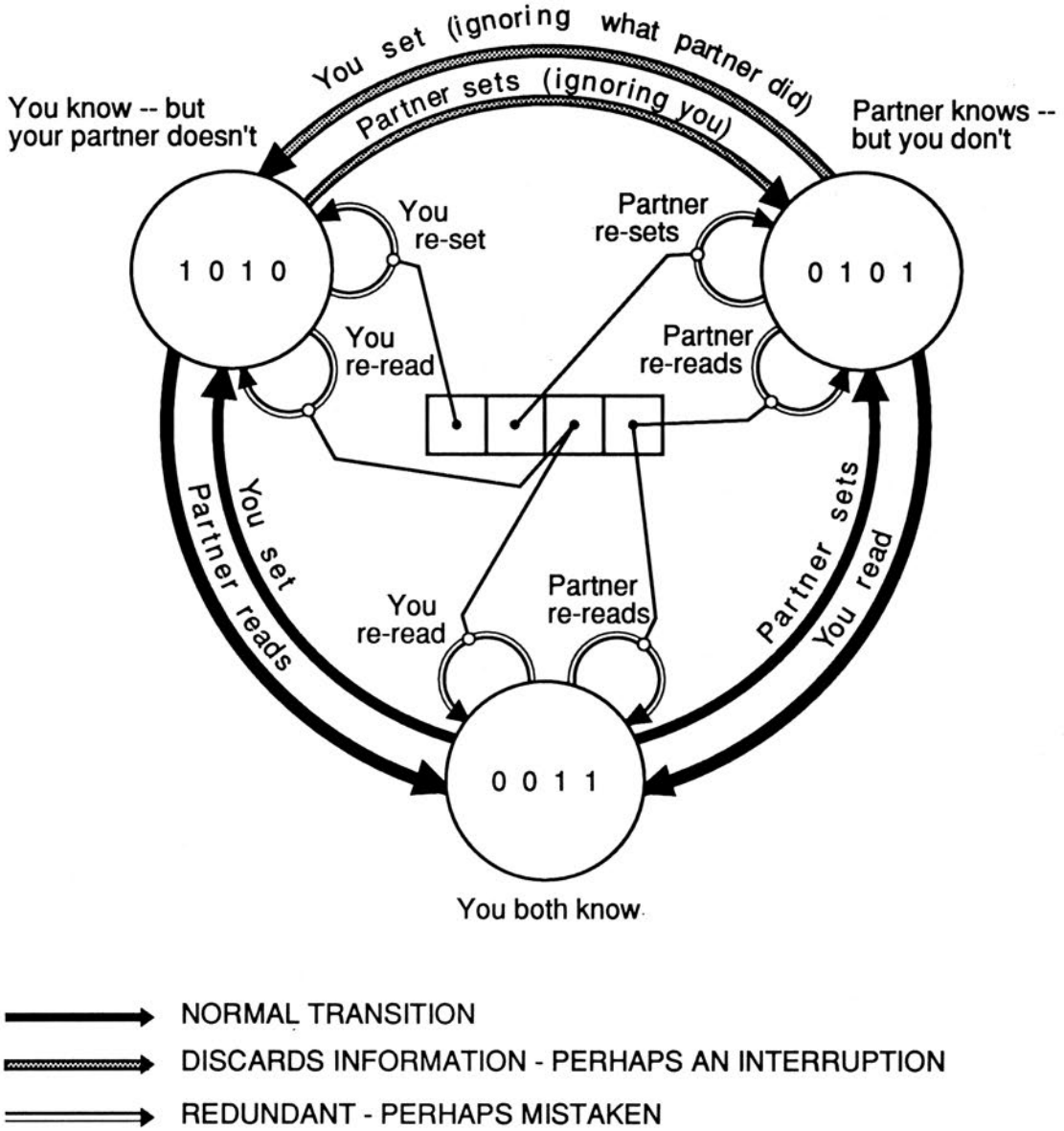


Figure 11–3 : Transition diagram for state changes.

The curved paths show the various ways you could get from one state to another. For example, if you are in the state shown by the lower circle ("You both know"), and then you set the shared name, the path labeled "You set" takes you to the state labeled "You know but your partner does not."

Normal orderly communication is represented by the pairs of solid lines linking the lower state ("You both know") to one or the other of the states at the top. A normal sequence is:

- You start in the lower circle.
- Then you *set* the shared name. That takes you to the upper left.
- Then your partner *uses* the shared name. That brings you back to the lower circle.
- Then your partner *sets* the shared name. That takes you to the upper right.
- Then you use the shared name. That brings you back to the lower circle again.
- And so on indefinitely...

The other paths represent various exceptional transitions. Mostly they arise from programming error, or from emergencies. Avoiding the erroneous paths is what the shared name controls are for.

Representation of Access Control

At the center of Figure 11–3 you see the *effective access control*, established by the way you and your partner have each set `⎕svc`. For each element of the access control, a line shows which paths are blocked by a value of `1`.

Exceptional Routes

Since there are two of you, and either of you could set the shared name or could use it, there are four arrows coming out from each circle in Figure 11–3. Some of the arrows do not go anywhere, but lead back to the same circle. That happens when either of you repeats an action – for example, if you already know the value of the shared name but you use it again anyway. These paths are shown by unshared lines. There are six of them altogether. At best, they are redundant. At worst, they may arise by mistake (for example, you *think* you are about to read new information, but in fact your partner has not yet sent any.) *Any of these six redundant paths can be blocked by an appropriately placed* `1` *in the access control.*

The two paths across the top are shown by shaded lines. They represent interruptions: when one of you ignores what the other has just set, and

instead sets a new value. When it is your turn to *use* the shared name, you have the right to *set* its value instead (ignoring the value presumably set there by your partner); this is not something you want to do all the time, but something that must be possible, as a way of signaling to a partner that you have to break out of the ordinary pattern of exchange. Neither of these paths can be inhibited. However, sending an interrupting message to your partner does not guarantee that the partner will notice it or heed it.

Attempting to Carry Out an Action That Is Inhibited

Suppose you have a name whose state at present is described by `1 0 1 0` ("You have set") and whose control has a `1` as its first element (to inhibit you from re-setting until your partner knows what the value is). You attempt to give the shared name a new value anyway. What happens? You experience a delay. There is no interrupt or error message, but SHARP APL waits to make your specification until your partner uses (or sets) the name you share. The delay lasts indefinitely – as long as it takes.

If you decide you would rather not wait, you can signal a *strong interrupt.* (You may want to do this if you suspect that there is something wrong with the AP, or that your attempt to re-set the shared name was ill-conceived.) This aborts your attempt to use the shared name. The interpreter signals event 1005, `sv interrupt`, and sets `⎕er` to contain the sentence it was attempting to execute at the time.

A similar delay occurs when a name's state has a `1` in the third position (meaning that you know what the name's value is), and there is a `1` in the third element of the name's access control (to inhibit you from re-reading until after your partner has set). There is an indefinite delay until your partner has again set a value for the shared name. As before, you can interrupt the attempt to use the interlocked name by signaling a strong interrupt.

Watch out: While you are trying to discover what went wrong with a program interrupted by an error, it may seem natural to attempt to display the current value of a name that is shared, just to take a look at what is going on. But such a display counts as a reference to the name, and might be inhibited.

Worse, you may thereby reference the value of the shared variable (having sent it to your terminal's display instead of to where it is needed), and thereby have disrupted the synchronization of the two tasks. Either of these may cause the application to fail.

When there is any possibility that you will need to re-examine the value a partner has set, it is essential that your first and only use of the shared

name is to assign its value to some other unshared name, so you can refer to it as often as need be.

Shared Names and ⎕pack, 4 ⎕ws and 6 ⎕ws

When you use a shared name in the argument of any of these three system verbs, as far as the Shared Variable Processor is concerned, this does *not* constitute a reference to the shared name. With `6 ⎕ws` and *monad* `⎕pack`, you get the last value visible in your workspace; with `4 ⎕ws` you get the size of the last value visible in your workspace. Thus, you only get a value that you *already know* (either because you have already made some other use of it, or because you are the one who set it). Using any of these three verbs has *no effect* on the shared name's state.

Which Actions Will Not Be Delayed?

By examining a name's state and its access control, you can tell which actions will *not* be subject to delay. Recall the four possible actions: you set, your partner sets, you use, and your partner uses. For a name α, an action you are at present free to take without being delayed[7] is any of those indicated by a `1` in the expression:

```
(⎕svs 'α') ∧ ⎕svc 'α'
```

Waiting for a State Change

Suppose you just want to wait until an AP accepts an offer. How can you say to APL, "Wake me when it answers"?

Or suppose you have a program that uses several shared names (for example, to communicate with two different auxiliary processors). The program may, on occasion, reach a point where it has nothing to do immediately, but is waiting for a reply to come in from one of its shared names – but cannot tell which one will be set next. How can you have the program *wait until something happens* that may require its attention?

You wait for a change in the state of your shared names by *using* the system noun `⎕sc`, which is automatically shared with the SHARP APL system. Whenever you use `⎕sc`, you get the value that the interpreter has set. That value is either `0` or `1`. The interesting property of `⎕sc` is usually not *what* its value is, but *when* you get it. While you are using shared names, between any two successive uses of `⎕sc`, the Shared Variable Processor must have detected a *state change*. If there has not been a state change, execution is delayed until there is one (or until a time limit that you have

[7] There is still the possibility that an action by your partner can change the situation between the time you execute this expression and the time you execute your action.

set expires). So when you receive your value of `⎕sc`, you are assured that something other than your task has done something that affects the shared names situation.

To set a time limit on the delay you experience, you set the value of `⎕sc` to the maximum number of seconds you are willing to wait. If no shared names event has occurred by the end of that time, the SVP sets `⎕sc` to `1` anyway. Thus, when you receive the value of `⎕sc`, you know that either there has been a state change or the time that you set has expired. It is then up to you to check the state of names that interest you to see what may have happened.

What Counts as a State Change?

A state change is an action (or several actions) that occurs outside your workspace but affects your use of shared names. State change is thus a rather general category. It includes:

- any new offer to share extended to you
- any *set* of a name you are already sharing
- any *set* of a shared name's access control
- any change in the *degree of coupling* of a name you have already offered
- the *first time* a partner *uses* a value that you have set
- expiration of the time limit you established by setting `⎕sc`
- the first shared variable operation you perform after a `)load` or `)clear` or sign-on
- a system shutdown warning post.

When you use `⎕sc` at a time when you have no valid clone ID, you receive the value `0`. Since you have to establish a clone ID before you can use a shared name (even if only by accepting the default clone ID), the value you receive is `1` whenever you are actually using shared names.

Once your task has established a valid clone ID, the Shared Variable Processor sets the value of `⎕sc` to `1` *each time there is a state change.* Each time you use `⎕sc`, you are getting a *new* value. The value is always `1`, but it is newly set there. It is the value that the SVP set *after* your previous use of `⎕sc`. The fact that you get a result indicates that there has been *at least one* state change since you last used `⎕sc`.

When a state change has occurred since you started sharing or since the last time you referred to `⎕sc`, you get a value when you use it. When there has been no state change, the interpreter does not return the value of `⎕sc`: it *waits until a state change occurs.* Hence, using `⎕sc` amounts to

waiting *indefinitely* or up to the time limit last set in `□sc` until some state change takes place. Note that if a potential partner retracts an offer that was made to you but which you have *not* accepted, that does *not* count as a state change.

Sequence for Using `□sc`

A reasonable sequence for using `□sc` is:

1. Using `□svs` (or also `□svq` if you expect new offers), verify that there is nothing you can do at present with the relevant shared names.
2. Set `□sc` to the maximum number of seconds you are willing to wait for a state change.
3. Use `□sc` in an expression such as `→(□sc/label),error`

   Provided you have a valid clone ID, that will delay the branch to `label` until some state change occurs. But if you lack a valid clone ID (and you cannot make effective use of shared names without one), control passes immediately to `error`.
4. When you have received a `1` from `□sc`, use `□svs` or `□svq` to review the situation and decide what action is possible or appropriate. (Note that there may be no detectable change in `□svs`, and the only appropriate action may be to repeat the sequence.)

Saving a Workspace Containing Shared Variables

When you save a workspace, SHARP APL requires that the value of each of its names must be physically present within it. When you give the command `)save`, you might be sharing a name whose value you have not yet used. (That is, your partner knows the value, but you do not.) If you were to use the name, the Shared Variable Processor would supply the value. But when you save the workspace, SHARP APL moves into the workspace the current value of each of the shared names to which you have not yet referred.

If it there is not enough space in the workspace to bring in the current value of the shared name, SHARP APL displays the message `ws full`, and does *not* save the workspace.

At a forced save (for example, at the end of an N- or B-task, or the save of `continue` at the involuntary end of a T-task), SHARP APL brings in the latest value of each shared name if it can, but goes ahead and saves anyway without the value of names for which there is no space.

12 *S-Tasks*

An S-task is an APL task controlled by way of a name shared with AP1, the APL S-task processor. The name "S-task" comes from *shared variable task.*

An S-task has no screen or keyboard; it is started and controlled not by a person but by another task.[1] Usually, it exists to provide some special service to the task that created it, or to a group of cooperating tasks.

The task that controls an S-task gives the S-task commands and receives the S-task's replies. The commands may be quite similar or even identical to those a human user would give a T-task. The principal difference is the way the controlling task sends commands and receives replies. This is illustrated schematically in Figure 12–1.[2]

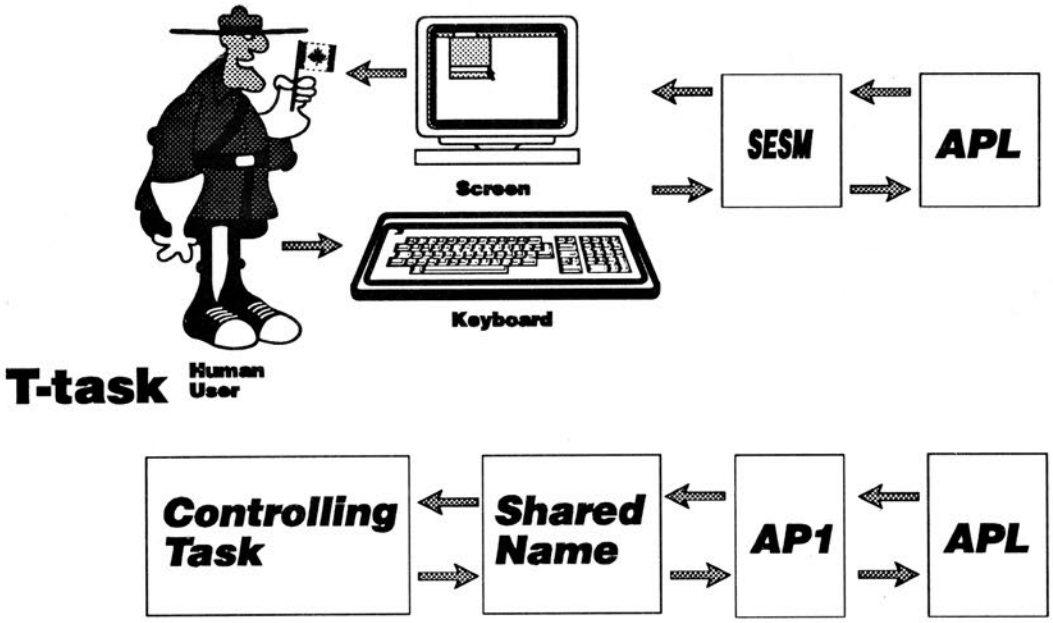


Figure 12–1 : Schematic representation of T- and S-tasks.

1 A task controlled from a terminal or keyboard is called a T-task. The human user who controls a T-task enters commands at the keyboard and receives replies at a screen or printer.

2 The session handler of a T-task, represented by the box labeled "SESM," may also be connected to AP1 and thence to SHARP APL, but this routing is invisible to the user at the terminal.

The controlling task formulates each command as a *character list.* It contains the same characters that a human user might enter at a keyboard. The controlling task passes its command to the S-task by assigning the character list to a name that it shares with AP1. AP1 sends the S-task's replies in the same way: it sets the shared name with a character list to represent the display.

Starting an S-Task

The controlling task initiates an S-task by offering to share a name with AP1 in a sentence such as:

```
1 □svo 'sh'
```

When AP1 accepts the offer to share, the controlling task sees the result of `□svo 'sh'` become `2`. AP1 is ready to start an S-task. As a first step, it requires the S-task to sign on, just as it would require a user from a terminal to sign on. AP1's first transmission to the controlling task is a *blot*, the standard signal to request a user number and password. (It is thus possible to start an S-task on a user number other than your own, but to do so you must have both the user number and the user's sign-on password.)

If the user number for the proposed S-task is the same as the user number that owns the controlling task, no password is necessary.

Assuming the user number and password are satisfactory, AP1 next sends its usual sign-on message (called the *banner*), just as it would to a newly signed-on terminal. Then it gives the task a *clear workspace*, or loads a `Profile` or `continue` workspace, as appropriate.[3]

Duration of an S-Task

An S-task continues until terminated, much as a T-task does. The controlling task may start an S-task to do a specific job, and when that job is complete, sign it off.

If the controlling task does not sign off the S-task it has started, the S-task may wait indefinitely (much as a T-task may wait while its human user sits at the keyboard but does nothing). However, as soon as the controlling task ceases to share a name with AP1, the S-task ends. Thus, ordinarily an S-task must end when the controlling task clears its active workspace, loads another workspace, or itself signs off.

An S-task can be made *freestanding*. A freestanding S-task continues to run after the controlling task retracts its offer to share with AP1. Usually, an

[3] For details of the start of a T-task, see Chapter 2, "Tasks, Users, Libraries".

S-task is left unattended in this manner while running a "server" program: that is, a program that waits for incoming offers to share with it, attends to them when they come, and then "sleeps" until something happens that requires it. Server programs are briefly discussed later in this chapter.

Once started, an S-task continues until one of the following occurs:

- The S-task terminates itself.
- The controlling task sends a sign-off command.
- The controlling task ceases to share the controlling name (by explicitly retracting it, expunging it, clearing the workspace in which it exists, and so on) *without* first notifying AP1 that the S-task is to be free-standing.
- An external command terminates it. This might be the system verb `⎕bounce`, or an action taken by the system operator.

A shutdown or crash of the SHARP APL system also ends an S-task. Although a B-task can be restarted following such a termination, there is no mechanism to restart an S-task.[4]

AP1 and Controlling Task Communication

The controlling task and AP1 share a name. Each sets values of that name. The values set by the controlling task are always characters, and always lists (that is, rank-1 arrays). The replies sent by AP1 are almost always character lists, too.[5]

The transmitted characters are APL characters, without translation (rather than the various transmission codes that may be used in communications between a terminal and a SHARP APL session handler).[6] However, AP1 is able to resolve overstrike sequences, so if the characters sent to it include (for example) the sequence ○ followed by *backspace* followed by |, it responds as if it had been sent the single character ⌽.

The reply returned *from* AP1 does not have a *newline* at the end. A line sent *to* AP1 does not need a *newline* at the end. AP1 considers a line sent to it up to but not including the first *newline* character, and discards any characters that follow the *newline*.

4 More or less continuous service can be provided by having each program that uses an S-task server check whether it is running, and if not, start it.

5 The only exception is that AP1 sends a numeric reply to a request for accounting information regarding the S-task.

6 In this regard, control of an S-task is simpler than transmission from the terminal of a T-task. Depending on the terminal, transmitting the characters `a←⌽⍳T` may require codes that change fonts, express ⌽ by transmitting ○–backspace–|, and so on. The session handler has to interpret those transmission codes as APL characters, but an S-task simply sends and receives the APL characters themselves.

Prefix Codes

Each time either side sets a new value for the shared name, the list that it sets consist of a *prefix code*, and then the characters of the command or reply.

For a transmission sent from the controlling task *to* AP1, the prefix consists of 4 characters. For a reply sent *from* AP1 to the controlling task, the prefix consists of at least 4 characters, but may be longer.

> ***Zero-Origin Indexing***
>
> SHARP APL uses index origin 1 unless you set it otherwise by `⎕io←0`. S-tasks make use of codes transmitted as arbitrary characters. Such characters are conveniently described by their 0-origin indexes in `⎕av`, so that the possible characters range from `⎕av[0]` to `⎕av[255]`. For that reason, ***0-origin indexing is assumed throughout this chapter.***

Prefix Codes Sent to AP1

For transmission *to* AP1, there are five possible prefix codes. One of these, the code for "APL input," precedes any line sent to the SHARP APL interpreter.

The other four codes stand alone. If you send one of the other codes as part of a longer line, the balance of the line is ignored.

The five prefix codes are as follows:

`⎕av[0 0 0 0]` ***APL input.*** The rest of the line is input to SHARP APL. This prefix is required for all APL input, including system commands, lines entered for immediate execution, lines in response to ⎕- or ⍞-input, lines sent in response to `⎕arbin`, and lines sent to the ∇-editor.

`⎕av[0 0 1 0]` ***Accounting request.*** In response to it, AP1 sets the shared name with a numeric list to report the task owner, the CPU units used and the task's elapsed time and CPU time (in the same way as `3↑⎕ai`).

`⎕av[0 0 2 0]` ***Interrupt.*** The effect of this signal is like signaling *Attention* from the keyboard: that is, it depends on the state of the S-task at the time. A single use may be a weak interrupt, and a second a strong interrupt. It may be interpreted as an input interrupt, file interrupt, and so on.

`⎕av[0 0 3 0]` ***Start freestanding.*** This code turns on the *independence* bit. It notifies AP1 that you permit the S-task to keep running after you retract the controlling name that you share with AP1. (See the discussion of "Freestanding S-Task," later in this chapter.)

`⎕av[0 0 4 0]` ***End freestanding.*** This code turns off the *independence* bit. It notifies AP1 that you assume responsibility for your S-task, so that it may survive only as long as you continue to share with AP1 the name by which you control it. (See the discussion of "Freestanding S-task".)

Prefix Codes Sent from AP1

Output from an S-task differs from input in two important respects:

- Whereas almost all forms of *input* are sent in the same way (with the prefix `⎕av[0 0 0 0]`), *output* from an S-task carries with it information about the *mode* in effect when it was sent and the mode that will govern interpretation of the next input. This information is carried in the fourth character of the prefix of each segment. *It is up to the controlling task to interpret it.*
- Whereas *input to* an S-task (like input to a T-task) is sent one line at a time, *output* from an S-task (like output for a T-task) may be spread over many lines. For this reason, output is segmented. Each value to which AP1 sets the shared name may contain not just one line of response, but many segments. Within the character list set by AP1, each segment starts with 2 characters indicating the length of the segment. A single setting of the shared name may contain only a single segment, or may consist of many segments.

  A complete reply from the S-task may span many sets of the shared name. It is up to the controlling task to unpack the segments within a single value of the shared name, and to read the shared name again when the prefix of the last unpacked segment indicates that there is more to come.

Segmented Transmissions from AP1

Each segment consists of 2 characters of length information, 4 characters of prefix code, and the text of the segment. The length is encoded by treating the first 2 characters as a 16-bit binary number. This value is the gross length, including the 2 characters of length information and 4 characters of prefix.[7]

[7] For the definition of a verb to unpack the segmented transmissions from AP1, see `unpack` in "Sample Listings of Programs Useful in Debugging an S-Task," later in this chapter.

Interpretation of Prefix Codes

`⎕av[0 0 1 0]` Your preceding entry was invalid.

`⎕av[0 0 2 0]` The APL interpreter lacked space to process your preceding entry.

`⎕av[0 0 3 0]` Although AP1 is active and able to reply, no S-task has been started and so no reply from APL is possible.

`⎕av[0 0 0,`*x*`]` Your preceding entry has been accepted. The value of the last character (here shown as *x*) describes the text that accompanies this reply and the S-task's present state.

Binary Representation of *prefix*[3]

Let the noun *mode* be defined thus:

```
mode←(8⍴2)⊤⎕av⍳prefix[3]
```

Then a 1 in each of the eight positions within *mode* has significance as follows:

mode[0] The S-task is now in immediate execution mode.

mode[1] The accompanying text is a blot (for instance, preparatory to sign-on).

mode[2] The S-task has not been started.

mode[3] The S-task is in *keyboard locked* state; in order to request input, you must send an interrupt signal. (This is useful for an S-task run from a device that uses a reversing-half-duplex line protocol.)

mode[4] An output interrupt has been received. You should reset the S-task's virtual cursor to the beginning of the next line. (This control is necessary because the buffering of transmissions between APL and the end user may leave the APL interpreter unable to predict the state of the device it is addressing.)

mode[5] Arbitrary I/O. This bit is used both to describe the output that accompanies the prefix and (where input is expected) to indicate that your next transmission will be interpreted as arbitrary input.

mode[6] The accompanying text is a prompt for input. This is your cue that it is your turn to send text to AP1.

`mode[7]` The accompanying text is output, and there is more output to follow (either a continuation of this display or the prompt for input). Read another segment; if necessary, use the shared name to receive more segments.

Common Values of `prefix[3]`

The list that follows shows the last character of the prefix both by `1 0⍕mode` and as an integer `2⊥mode`.

`00000001`	1	Normal output, more to come.
`00000010`	2	Normal output, final (prompt for ⍞-input or ∇-editor input).
`00000100`	4	`⎕arbout` or `⎕arbin`, more to come.[8]
`00000101`	5	`⎕arbout`, final.
`00000110`	6	The accompanying text is a `⎕arbout` prompt for `⎕arbin` input.
`00001001`	9	APL requests the controlling task to reset the position of its cursor, following an interruption of output.
`00010001`	17	The S-task is ready for input, but its keyboard is in the locked state, awaiting an interrupt signal.
`00100000`	32	The S-task is not signed on to APL.
`01000001`	65	Part of a blot; there is more output to come.
`10000010`	130	Prompt for immediate-execution input. This is the standard prompt for a new input from the controlling task when APL has completed execution of the line previously sent to it.
`11000010`	194	Blot prompt for immediate-execution input.
`11100010`	226	Sign-on blot prompt.

Freestanding S-Task

It is possible for the controlling task to retract the offer to share the controlling name with AP1. When it does so, the S-task is left with no means of control, and is said to be *freestanding*.

[8] Codes 4 and 5 are indeed defined and used thus, despite the fact that in general bit 7 means "more to come." In most contexts (but not here) and odd value means there is more to come, and an even value indicates that this is the last of an output sequence.

A freestanding S-task may or may not continue to run. Its fate depends on the current value of the *independence* bit which AP1 uses for each S-task. The controlling task turns on the S-task's independence bit by the transmission `⎕av[0 0 3 0]`, and turns it off by the transmission `⎕av[0 0 4 0]`.

The independence bit affects the S-task as follows:

On ***Freestanding.*** The S-task continues indefinitely, executing the last instruction given to it while it was in immediate execution mode. When it completes execution (and therefore returns to immediate execution for a new instruction), it terminates. Its active workspace (unless cleared) is saved under the name `continue`, or under the name specified by the system noun `⎕twsid`.

A freestanding S-task provides a service equivalent to that of an N-task. Note that the instruction the S-task is executing may be a program that loops indefinitely, perhaps causing the S-task to act as a resident server which runs until some prearranged criterion is met, or until you terminate it from outside.

Off ***Not freestanding.*** The S-task continues indefinitely as long as the controlling task continues to share the name with AP1. The S-task may return to immediate execution mode (awaiting a new instruction). While the controlling task sends it no new instruction, it simply waits, inactive, until it receives one. When the controlling task retracts the name it shares with AP1 (for any reason, by any means), AP1 immediately terminates the S-task, saving its active workspace (unless cleared) under the name `continue`, or under the name specified by the system noun `⎕twsid`.

Default Value of Independence Bit

SHARP APL assumes that by default the independence bit is *off*. A freestanding S-task cannot survive unless you first explicitly set the independence bit *on*.

Reconnecting to a Freestanding S-Task

You may need to reconnect to an S-task when the S-task is still running but you are no longer sharing with AP1 the name formerly used to control it.

To reconnect, you must have the same user number and clone ID as you had when you started the S-task, and must offer the same name. When AP1 accepts the share, you are again connected to the S-task.

If – as might well be the case if the S-task is a server – you left the task running an endless program that waits until it receives a request, you may have to transmit an interrupt before you regain direct control of it.

Sample Programs Useful in Debugging S-Tasks

The following programs start an S-task and exchange commands with it.

All references to the shared name are handled by the verbs *svset* or *svuse*. As a side effect, they keep a log of each transaction as a two-column boxed table, with the prefix codes in the left column (as integers) and the transactions in the right.

These verbs make no assumption about what the shared name is, but accept it as a character argument. To transmit the command ⍳5 to AP1 by way of the shared name *a*, you execute:

```
log←2 0⍴0
'a' svset ⎕av[0 0 0 0],'⍳5'
```

To receive AP1's reply, you execute:

```
svuse 'a'
```

Their definitions are:

```
   ∇ svname svset α
[1]  ⍎svname,'←α'          ⍝Set the shared name
[2]  log←log⍪(⎕av⍳4↑α)⊃4↓α ⍝Keep record in log
   ∇

   ∇ z←svuse svname
[1]  z←⍎svname             ⍝Use the shared name
[2]  →(0=1↑0⍴z)/num        ⍝Test for numeric return (accounting)
[3]  log←log⍪unpack z      ⍝Interpret and log character return
[4]  →0
[5] num: log←log⍪0⊃z       ⍝Log numeric return
   ∇
```

To start a new S-task, using the name specified in the left argument and the user number in the right, execute:

```
    ∇ z←name sstart n;p;⎕io
[1]   ⎕io←0
[2]   z←1 share name          ⍝Offer name to AP1
[3]   p←GetPassWord n ⍝Get password if not your user number
[4]   name svset ⎕av[4⍴0],')',(⍕n),':',p
[5]   z←unpack svuse name     ⍝Get AP1's reply
[6]   display z               ⍝Show the sign-on message
[7]   z←1 2 ⍴n,findTaskNo z ⍝Get task's user, task no.
    ∇
```

The verb *GetPassWord* (not shown) returns an empty list when the user number for the proposed S-task matches that of the controlling task, but otherwise prompts for a password to be supplied. The verb *findTaskNo* (not shown) extracts the new S-task's task number from the sign-on message, and returns it and the S-task's owner as a numeric result.

The verb *unpack* decodes the transmission from AP1:

```
    ∇ z←unpack ⍵;s;l;header;text;⎕io
[1]   z←0 2⍴⎕io←0          ⍝Return 2-column array of boxes
[2] test:→(0∈⍴⍵)/0         ⍝ from segments sent by AP1.
[3]   l←256⊥⎕av⍳2↑⍵        ⍝1st 2 characters are segment length
[4]   header←⎕av⍳4↑2↓⍵     ⍝Header is next 4 characters
[5]   text←6↓l↑⍵           ⍝Text is what's left of the gross length
[6]   z←z⍪header⊃text
[7]   ⍵←l↓⍵
[8]   →test
    ∇
```

The verb *display* assembles the returned text at the terminal. Its only choice is between ordinary and arbitrary output:

```
   ∇ display ω;⎕io;i;code;text;mode
[1]   ⎕io←0                    ⍝ω[;0] is boxed prefix codes
[2]  test:→(0∈⍴ω)/0            ⍝ω[;1] is boxed text
[3]   code←>ω[0;0]
[4]   mode←(8⍴2)⊤code[3]       ⍝See "Binary Representation
[5]   text←>ω[0;1]             ⍝of prefix[3]"
[5]   →mode[5]/arb
[6]   ⍞←text                   ⍝No newline is added
[7]   →end
[8]  arb: ⎕arbout text
[9]  end: ω← 1 0 ↓ω
[10]   →test
   ∇
```

Sample Programs for a Server for Multiple Users

The following definitions are intended to illustrate how you might set up a task that can accept offers to share with it and can handle work in rotation from each of the various sharers, each running at its own pace.

The overall manager is called *monitor*. It uses an internal table called *names* which contains unique arbitrary names that it will assign to those who share with it, and a Boolean called *inuse*, which marks those names which are currently active. These are initialized in lines 2 and 3. As soon as it is started, *monitor* runs once through its cycle (probably finding nothing to do) and then sleeps (line 8) until a shared name event occurs. When it detects an event, it:

- executes any new commands sent to it (*work*)
- checks that those formerly sharing are still there, and deletes those that are not from its table of users (*oldoffers*)
- checks for new offers to share, and adds them to its table of users (*newoffers*).

```
   ∇ monitor cloneID
[1]   ⎕signal (cloneID≠⎕svn cloneID)/555
[2]   shared←0 3↓usage←0 4⍴IDs←0 2⍴0
[3]   inuse←(1↑⍴names)⍴0          ⍝names: internal substitutes
[4]  loop:work(⎕svs shared)[;2]   ⍝Do names that were set
[5]   oldoffers 2=⎕svo shared     ⍝Purge retractions
[6]   newoffers ⎕svq ⍳0           ⍝Do new offers; call work
[7]   →(0∈⍴shared)/0              ⍝Quit if no one to serve
[8]   →⎕sc/loop                   ⍝Wait for work to come
[9]   ⎕signal 556                 ⍝Invalid clone ID
   ∇
```

The principal verbs used by *monitor* are *work*, *oldoffers*, *newoffers*, and *picknames*.

For each of the shared names that has been set by one of the monitor's users, *work* handles the instruction it has received by passing it to a verb called *process*, whose left argument identifies the user and whose right argument is the user's request:

```
   ∇ work set; shv; user; n; Δ; i; ⎕trap
[1]  →(set∧.=0)/i←0          ⍝Boolean: which have been set
[2]  ⎕trap←'⍛ 6 c →test'     ⍝If offered name had no value
[3]  n←⍴Δ←set/⍳⍴set          ⍝Names whose values changed
[4] test: →(n<i←i+1)↑0
[5]  shv←shared[Δ[i];]       ⍝Pick one shared name
[6]  user←IDs[Δ[i];]         ⍝ and its owner
[7]  ⍎shv,'←user process ',shv ⍝use and set the name
[8]  →test
   ∇
```

The verb *oldoffers* purges the table of shared names of names no longer needed (because the tasks that offered them have withdrawn their offers):

```
   ∇ oldoffers OK
[1]  →(∧/OK)/0
[2]  ⊣⎕ex (~OK)⌿shared          ⍝Retract unneeded names
[4]  shared←OK⌿shared           ⍝Purge shared
[5]  IDs←OK⌿IDs                 ⍝Purge IDs
[6]  inuse←∨/names∧.=⍉shared    ⍝Purge inuse
   ∇
```

The verb *newoffers* checks each task ID that has made a new offer, assigns an internal name to it, and responds with an offer of its own. Then it adds the new name to its table of shared names, and similarly amends its list of users and the Boolean *inuse*:

```
   ∇ newoffers A; ID; new; surrog; i; n; rn
[1]  →(0=n←⍴A)/i←0          ⍝ A has IDs from ⎕svq ⍳0
[2] test: →(n<i←i+1)/end
[3]  ID←A[i;]
[4]  rn←1↑⍴surrog←⎕svq ID   ⍝Number of names offered
[5]  new←rn picknames ~inuse ⍝Update inuse
[6]  ID←((1↑⍴new),2)⍴ID
[7]  ⊣ ID ⎕svo new,' ',surrog
[8]  ⊣ 0 0 1 1 ⎕svc new     ⍝User falls through if I die
[9]  shared←shared,new      ⍝Update shared
[10] IDs←IDs,ID             ⍝Update IDs
[11]  →test
[12]end:                    ⍝If user set before offering
[13] work(-1↑⍴shared)↑(1↑⍴new)⍴1
   ∇
```

A verb *picknames* makes sure that monitor has distinct names, regardless of the names that the tasks offer to it:

```
   ∇ z←n picknames free;loc
[1]  loc←n⍴free/⍳1↑⍴names
[2]  inuse[loc]←1
[3]  z←names[loc;]
   ∇
```

13 *Batch Tasks*

In addition to *interactive* tasks (which are directly controlled from a keyboard or through a shared name) SHARP APL also provides batch tasks. They run autonomously, unsupervised and unattended.

A batch task has no ongoing interactive control. It receives its instructions at the moment it is launched. After that, it progresses on its way until it completes what it started to do – or fails due to an error or interruption. This chapter is about batch tasks and how you prepare and start them.

Everything a batch task does must proceed from its initial instruction. In the commonest form of batch task, the principal information required is the name of a workspace. When the SHARP APL system has loaded that workspace, it executes the workspace's latent expression, `⎕lx`. The latent expression in turn invokes a program stored in the workspace. When SHARP APL completes execution of the latent expression, the task ends.

The other form of batch task starts by cloning the current active workspace, and continues executing whatever the active workspace was executing; see "Split workspace", below. It too runs only until SHARP APL completes execution of that program, and it ends when the program's execution halts.

A batch task is never in immediate-execution mode. As soon as its initial instruction is completed (or it encounters an untrapped error), it ends.

Although a batch task cannot get new instructions as a T-task or an S-task does, it may nevertheless use files that other tasks are using, or share names with other tasks. In those ways it may acquire new information that modifies its course. It can continue to exchange information with its parent task, perhaps looping and waiting for the parent task to send new orders. It may accept shared name offers from many users, and thereby serve as way of coordinating the access of many tasks to some common resource. Such a task might run indefinitely, "sleeping" until one of the names it shares conveys a new instruction. But if its execution ever halts – the point at which an interactive task would return to immediate execution to wait for new instructions – a batch task terminates.

How Batch Tasks Are Started and Controlled

There are two types of batch task: N-tasks and B-tasks. The main difference between them is *who controls when they start.*

N-task An N-task may be launched by an already-running task as a side-effect of the system verb `□run`. Such a task starts at once, as soon you execute `□run`. The argument of `□run` supplies what the task needs to know (see "Starting an N- or B-Task," below).

An N-task may also be started at a specified future time by submitting a request to the Batch Task Scheduler (see "Submitting a B-Task Request," below).

B-task A B-task is run on your behalf by a SHARP APL facility called the *Batch Task Scheduler*. You send the Scheduler a request that contains the same information that would be needed to start an N-task, along with additional information regarding *when* or under what conditions you want it to be run. At some later time – perhaps hours or days later – the Batch Task Scheduler runs the task for you.

Once an N-task or B-task has started, you have almost no further direct control over what it does. However, using the system verb `□runs` (from another task), you can see that a batch task is running and how much elapsed or CPU time it has consumed. (Of course, for a B-task you do not know exactly when the Scheduler will run it, so you may not be around when it runs.) The only direct control you retain over a batch task once it starts is the system verb `□bounce`. Using `□bounce`, you can terminate a task that you started (including an N-task) or a task that is run on your user number (including a B-task). This is principally an emergency measure, to intervene with a task that seems to be running away or to be endlessly waiting (for example, for an shared name event).

Alternative Starting Points for an N-Task

To start an N-task, you identify the workspace with which the new task will start. There are two ways to do this, identified as *autostart* and *split workspace*.

Autostart You provide the name of a saved workspace. When the system starts your N-task, it loads the workspace you have indicated. Then it executes the saved workspace's latent expression.

Split workspace The system starts your N-task by cloning your present active workspace. The new task continues executing exactly the same program as your current active workspace. There is no loading of a saved workspace, and hence no latent expression. However, the spawned N-task will be in a somewhat different state from its parent, as it will

not have any files tied, nor will it have any shared names outstanding.

The Owner of a Batch Task

The owner of a task is the user identified in `1↑⎕ai`. The owner's user number is charged for the task, and the owner's user number determines the limits, quotas, or access rights that apply to the task. Any reference to a file or workspace that does not explicitly mention the library is understood to refer to the owner's library.

Any task can start an N-task or submit a request for a B-task. Ordinarily, the user whose task started the batch task is the task's owner. However, it is possible to start a task on behalf of another user. In that case, the user that owns the task is different from the user that started it. To submit a task on behalf of another user, you have to provide both the owner's user number and the owner's password. Thus starting a task on someone else's user number is rather like signing on as that person.

For every task (both interactive and batch) the following are established when the task starts:

- ***Task number.*** This is a unique number, assigned by the system at the moment the task starts. (Task numbers are recycled after task 9999.)
- ***Owner.*** The user number that owns the task.

These numbers are described in the first two columns of the result of `⎕runs` (described in Chapter 16, "System Nouns and Verbs"). The system noun `⎕runs` describes all the tasks that are now running for which you are either owner or starter.

Limits on Type and Number of Tasks

Each SHARP APL installation sets a limit on the total number of tasks (of all types) that it can handle simultaneously. It may also set a maximum on the total number of tasks owned by the same user, or the total number of tasks of a particular type that a user number may have. The Batch Task Scheduler has its own limit to the total number of tasks it can enqueue, and the total number of requests owned by the same user.

Restart After Interrupted Work

A task started by the Batch Task Scheduler can be automatically restarted (and its work continued) after being interrupted by a system shutdown or a system crash. To permit this:

- You must specify one of the possible *restart* options at the time you submit the request.[1]
- You must construct the programs that the task will run in such a way that they can be resumed after an interruption.

When a task is interrupted by a normal shutdown, the system signals event 1002, `Attention` . Work is halted at the beginning of the next line to be executed, and the active workspace of an N- or B-task is saved under then name provided when it was started.

When a task is interrupted by a system crash, work may be interrupted at mid-line. It may or may not be possible for the system to recover the active workspace. Even when the active workspace is recovered, it may be from a time a few seconds earlier than the most recent file updates, so there is a risk the file operations are out of synchronization with the recovered workspace. Nevertheless, it is possible to design a task so that it can be restarted after a crash, either by starting again from the original workspace, or by starting from the recovered workspace (if it exists). Such a restart is attempted only when the request to the Scheduler permits *restart after an abnormal end* (a phrase usually shortened to *abnormal restart*). The usual technique for abnormal restart is to rely on files that keep track of the task's progress, and have the task start by checking the state of its files before proceeding.[2]

Pricing and Priorities

The SHARP APL system can set different dispatching priorities for different kinds of tasks, and permits separate accounting of charges accrued during them. The installation may set different charges and dispatching priorities for N- and B-tasks than for T- or S-tasks.

Starting an N- or B-Task

To start a batch task, there are five items you must state. For an N-task, these appear directly in the argument of `□run`. For a B-task, the same information is provided in the request to the Batch Task Scheduler, along with some additional information that applies only to B-tasks.

The argument of `□run` is a character list consisting of five variable length fields separated by blanks. The same fields appear as part of the argument

1 See the separate publication *Batch Task User's Guide* for details.

2 Restartability is discussed under the heading *Starting – and re-starting – a Batch Task* in the separate publication *Batch Task User's Guide*.

to the verb that submits a B-task (along with some additional specifications specific to a B-task). The five fields are:

Owner The user number and sign-on password of the account that is to own the task, separated by a colon. You need specify this information only when the owner will be an user other than your own. However, *the colon is required.* If you omit an user number or password, you must still include a colon.

For a split-workspace N-task, this field must contain only a colon.

Initial workspace The name of the workspace the must be loaded to start the task. When the workspace belongs to a user other than the owner of the proposed task, preceding the workspace name you must show its library number, separated from the workspace name by blanks. When the workspace is locked, you must follow its name by a colon and its password.

For a split-workspace N-task, this field must contain only a colon.

Save-name The name under which the task's active workspace will be saved when the task ends. You *must* supply a name here. There is no need to supply a user number with the name: the workspace can only be saved into the library of the task's owner. Moreover, the owner's quota of saved workspaces must be sufficient to permit saving the workspace. The save-name can be changed by the task by using the system verb `⎕twsid`.

Watch out: When the task ends (unless it uses the trap `2001 d clear`) its active workspace will be saved under the name you supply here *even if a workspace with that name already exists*. The workspace saved at the task's end may thus overwrite an unrelated existing workspace.

Max CPU Maximum number of CPU units permitted for this task. SHARP APL checks the actual CPU total for the task against this maximum periodically. When it finds that the actual has exceeded the limit, it halts execution and saves the workspace. (Because of the way the check is processed, the actual number of CPU units a task uses may slightly exceed the maximum specified in the argument.)

Do not set this limit unrealistically. If you set the limit too low, your task will be interrupted before it can finish.

If you set it too high, a programming error may run up a large bill before work is interrupted. The system interprets a `0` here to mean "no limit." This is permissible *but very dangerous*. An endless loop could run up an enormous bill before someone stops it.

Max elapsed Maximum elapsed time, in seconds, permitted for this task. When the SHARP APL system's supervisor finds that the actual elapsed time has exceeded this limit, it halts execution and saves the workspace.

Here again, you should not set this limit unrealistically. If you set the limit too low, your task will be interrupted before it can finish. If you set it too high, a programming error can run unchecked until work is at last interrupted.

For a B-task, an excessive time limit may cause the Scheduler to delay running your task because it appears more demanding than it really is. The Scheduler will start your B-task only at a time when the interval you requested appears to be available. However, if you permit the Scheduler to interrupt your task and restart it later, the Scheduler can fit it in more easily; see the discussion of the *restart* specification in the separate publication *Batch Task User's Guide*. It is permissible (but dangerous) to set *max elapsed* to 0 (meaning "no limit"). A B-task whose time limit is 0 *must* permit interruption and restart after system shutdown.

Sample Arguments for □*run*

```
□run ': MyWS End 10 600'
```

Owner: You (because the colon alone means none is stated).

Initial workspace: Your workspace called *MyWS*.

Save-name: A workspace in your library, to be called *End*.

Max CPU: 10 CPU units.

Max seconds: 600 elapsed seconds (that is, 10 minutes).

```
□run ': 75461 Acct:gastos Final:gh2Q2 200 8000'
```

Owner: You (colon alone means no owner is stated).

Initial workspace: From library `75461`, the workspace named `Acct` with password `gastos`.

Save-name: A workspace in your library, to be called `Final` and locked with the password `gh2Q2`.

Max CPU: 200 CPU units.

Max seconds: 8000 elapsed seconds.

```
□run ': : end 80 1800'
```

Owner: You.

Initial workspace: Your active workspace. This invocation is called a split-workspace N-task.

Save-name: A workspace in your library, to be called `end`.

Max CPU: 80 CPU units.

Max seconds: 1800 elapsed seconds.

Result of □run

The result of the system verb □run is a two-item numeric list indicating the outcome of your instruction to start an N-task. When the first item is `0`, the N-task was successfully started. The second item is usually the task number of the new task. In the case of a split-workspace □run, the task number observed by the spawned N-task is `0`. This allows the parent task and the spawned task to distinguish themselves from one another.

When the first item is *not* `0`, the N-task was not started, and the second item of the result is an error code indicating why not. Since a B-task may encounter the same problems as an N-task (as well as some others of its own) the same error codes also occur in the report on the status of a B-task request.

For a list of the error returns from □run, see the description of □run in Chapter 16, "System Nouns and Verbs".

Distinguishing the N-Task from Its Parent

You start a split-workspace N-task by cloning your present active workspace. Since your active workspace is split into two identical workspaces, they need a way to distinguish which is which. They are distinguished by

the value in the second item in the result of □*run*. (Since the N-task was successfully started, the first item in the result is the number 0.)

In the parent workspace: The second item is the task number of the N-task.

In the N-task thus started: The second item is 0.

Thus it is essential for the program that launches a split-workspace N-task to include a test something like this:

```
[ ]  r←□run ': : ',endws,maxCPU,maxSecs
[ ]  □signal (0≠1↑r)/Nfailed
[ ]  →(0 0≡r)/Ntask ⍝Test: "Am I N-task or parent?"
[ ] Parent: ...
[ ]  ...
[ ]  →end
[ ]  Ntask: ...
```

Reporting on the Status of an N-Task

The value of the system noun □*runs* is a table that reports the status of all active tasks of which you are the owner or originator. For each task, it indicates the task number, owner user number, initiator user number, task type, CPU time, elapsed time, CPU time limit and connect time limit. □*runs* includes all types of tasks.

□*runs* is particularly useful for checking whether an N-task is running and, if so, its current elapsed time and CPU time. See the description of □*runs* in Chapter 16, "System Nouns and Verbs".

Submitting a B-Task Request

You submit a request for a B-task by using verbs from a public workspace provided with the APL system, typically called 1 *btaskreq*. Verbs defined in the workspace accept as argument a list of characters describing the task to be run in the same way as an N-task. At the time you provide additional information regarding the time or date that the task can be run, prerequisites, restartability, and so on. These are described in the separate publication *Batch Task User's Guide*.

14 *APL Component Files*

A file is a collection of data outside the APL workspace but accessible to an APL expression executed from within the workspace. SHARP APL provides access to files in two ways:

- to standard *MVS files*, by way of auxiliary processors, such as TSIO, SSQL, and VSDI
- to *APL component files*, by way of APL system verbs provided for that purpose.

The APL component file system provides concurrent indexed access to variable length self-describing APL nouns.

The sections that follow explain the preceding sentence, and contrast the APL file system to the "native" files supported by MVS. If these concepts are familiar, you may want to skip forward to "Concepts of APL Component Files," later in this chapter.

Concepts Relevant to Files in General

A file is a collection of data identified by a single name.[1]

For most purposes, a file is not processed as a whole, but as a series of segments. Files differ widely in the manner in which you specify the particular portion you want and in the manner in which the computer interprets the data it finds in such a segment.

The conventions of the APL component file system differ in several respects from those that apply to other MVS files. To explain the difference in approach, the paragraphs that follow first summarize some characteristics of MVS files.

Opening a File

In order for a program to make use of a file, the program must first request the operating system to *open* it. Opening a file causes the operating system to locate the file and to maintain information that will be used during references to it. In addition, the operating system establishes the format in

[1] At least conceptually, a file is stored together (for example, along a tape, or in a series of tracks in a disk). However, quite often the physical storage is fragmented; although it appears to you as a continuous stream of data, it may in fact be stored at different physical locations.

which the data is stored, and rules for the format in which it is exchanged with programs that use it. When you ask the operating system to open a file, you indicate the type of access you want (for example, read-write, read-only). Opening a file for *exclusive use* causes the operating system to reject similar requests from others.

In most situations, you identify a file by name only at the time that you open it. Thereafter you refer to it by some sort of shorthand based on the way you opened it.

Records

An MVS file is divided into *records*. A record is a block of data written or read as a unit. A file is usually open-ended, so that it would not be practical to read all of it at once. Data is transferred a record at a time, or sometimes in a block of consecutive records.

File Format

Each MVS file has a *data control block* (DCB), which describes the format of the records in it. A file usually has a single DCB: the *same* format applies to *all* records. The DCB indicates (among other things) whether the records have the same length or may vary in length.

Access to Individual Records

There are several different ways to access a particular record within a file. These include:

Sequential access When you start reading a file, you get its first record. Thereafter, each time you read you get the next one, until you reach the end.

Indexed access You indicate the record number of the record you want. The operating system calculates where to find that record, and returns it. Indexed access may be available only to a file in which all records have the same length.

Keyed access You indicate a set of characters (the "key") that identify the record you want, and the operating system finds and returns the record having that key.

Each Record is Interpreted According to the File's Format

The access services provided either by the operating system or by the program interpret the data within each record according to the file's DCB. For

example, the first 20 bytes might be interpreted as letters of the alphabet, the next 16 bytes as four integers, the next 80 bytes as a 2-by-5 table of floating-point numbers, and so on. When a program running under MVS reads such a record, it is customary to specify names for the various objects within a record; the act of reading a record in effect supplies a value for each of the names.

However, when you use TSIO to read from an an MVS file into an APL active workspace, each record is returned as a *single* object. It may appear as a string of characters, or as a string of numbers, depending on the command you give TSIO when you open the file. The task of interpreting the characters, or of partitioning the record into segments that have different formats, or assigning names to the various segments, are all the responsibility of the APL program. Because there are various different systems for representing numbers, or of assigning binary codes to characters, reading a record implies some sort of translation between the encoding used in the MVS file and the encoding used by the APL system. TSIO permits you to specify one of several ways of translating characters or of treating clusters of bytes as numbers.

Concepts of APL Component Files

An APL component file differs both from ordinary MVS files and from an APL workspace. The points of difference are listed in what follows.

Ownership

A file is owned by a user enrolled in the SHARP APL system. The set of files owned by a user includes the set called the user's *file library* (in the same way that set of saved workspaces owned by a user is called that user's *workspace library*), as well as any files which exist in other (public) libraries.

The full name of an APL file consists of the characters representing the library number (up to 10 digits), a blank, and the characters of the name (up to 11 characters). An APL task can refer to files in its own file library without stating the library number explicitly.

Organization into Libraries

An APL file is stored in a *file library* associated with a user number. This is done in the same way that APL workspaces are grouped into libraries.

From the user's standpoint, APL files are *not* further organized into "directories," "volumes," etc., in the way that MVS files are organized.[2]

SHARP APL provides facilities to let you grant access to other users to your files and libraries. See "File Access Controls" and "Library Access Controls," later in this chapter.

Limits and Quotas

Each time an APL file is created, you specify a maximum size for it. This is called a file *reservation.* Once the file's size exceeds its reservation, SHARP APL no longer permits you to append further components or to replace existing components with larger ones.[3]

The reservation is not an absolute limit. You can "resize" a file (that is, set a different maximum) at will, or as the need arises (perhaps automatically, by trapping the error event `file full`; (see Chapter 10, "Event Handling".) The new size may not be smaller than the file's actual size. Having a file reservation serves primarily to guard against inadvertent growth.

The APL system's accounting algorithms keep track of file space actually used, not the amount reserved (which may be far greater). Because storage is allocated in units of physical records, the amount of file space consumed will appear to increase in steps, rather than smoothly.

For each user, the APL system also sets two *file quotas.* One specifies the maximum number of files that a user may own. The other specifies a maximum for the sum of all the user's file reservations. These file quotas can be set only by the system administrators, usually upon request.

Components

An APL file is made up of components. A component roughly corresponds to a *record* of an ordinary MVS file in that the APL file system transfers data between a file and the active workspace one component at a time.

Each component of an APL file is a noun. Like a noun in a workspace, a component may be an array of any rank (item, list, table, higher-dimensional array); its data may be of any type (numeric, character, boxed, or package). A file may contain the definition of a user-defined verb either as its character representation or as the referent of a name in a package.

---

[2] Of course, the APL component file system is ultimately built upon the tools provided by the MVS operating system, but that is not visible to the APL user.

[3] However, while the file is smaller than its reservation, SHARP APL will accept a change that takes the file past its reservation.

Indexed Access

A component of a file is identified by its *component number*. Component numbers are consecutive positive integers, ranging from the number of the first component up to the number of the last component. Components are numbered in 1-origin, so that the lowest possible component number is component 1.[4]

A component does not have a name. The noun in a component may be assigned to a name when it is transferred to the active workspace of an APL task.

Component Information

The file system automatically records for every component of every file the *size* of the component (in bytes), the *user number* of the task that updated it, and the *time* at which that update occurred. This information can be extracted by the system verb `⎕rdci` (see Chapter 16, "System Nouns and Verbs").

Self-Describing Data

An APL noun is *self-describing*. That is, in an APL workspace, each noun contains a statement of its type, rank, shape, and so on. This descriptive information is also stored with each component of an APL file. Thus the interpretation of a component depends on the data description stored as part of each component (rather than on a data control block (DCB) that applies to all records of an MVS file).

Variable-Size Components

Each component may be of any size. Moreover, when you replace the noun stored in a particular component, the replacement may also be of any size. There is no requirement that the replacement match the shape of the noun stored in that component earlier; the replacement may be smaller or larger.

When you replace the noun stored in a component with a noun that takes *more* space, the file system (although preserving the noun's component number) writes the new value in a new location. It may or may not be able to find an immediate use for the space the component formerly occupied. Similarly, when you replace a noun with one that takes *less* space, the file system puts the new value where the old one was, but may be unable to find an immediate use for the unused space. During the site's periodic

[4] The first component is initially component `1`, unless you specify otherwise in `⎕create`. However, if you drop some components from the beginning of a file, those that remain are not renumbered, so the first remaining component will then have a number higher than `1`.

maintenance and backup, files are completely rewritten and defragmented, and these problems disappear. But less-than-optimum storage (reflected in a gross file size that is greater than the sum of the sizes of the components) remains evident until maintenance is performed. Thus, file sizes may change without any alteration by a user.

Open-Ended Size

A file may be of any size. Data is transferred between file and workspace one component at a time (so a component may not be larger than the workspace that created the file). There is no limit to the total size of a file other than that imposed by the physical limits of the system that contains it, and administrative limits imposed on the number and size of files a particular user may own.

A common use of files is to partition data into components that may be processed serially, to reduce the amount of data in the workspace at a given moment. In similar fashion, a large application may partition the set of definitions it uses into those required at different phases of the work. A master program reads in the definitions needed for a particular phase, and then, when they are no longer required, replaces them with those needed for the next phase.

File Primitives

An APL component file is manipulated by primitives that you can enter at the keyboard or execute from within an APL program. Each of the file primitives is a *system verb*, written with a name whose first character is □. They are summarized in this chapter and described individually in Chapter 16, "System Nouns and Verbs".

Tying Links a File to the Verbs that Use It

To make use of a file, an APL task must first *tie* it. Tying a file is analogous to opening an MVS file. The tie verbs `□tie` and `□stie` pair the name of a file with an arbitrary positive integer called the *tie number*. The number is arbitrary; it may be any number that your task is not already using as a tie number. While the file remains tied, you identify the file *not* by its name but by its tie number.[5]

The verb `□tie` requests *exclusive* use of the file; while you have it exclusively tied, no other task can tie it. The variant `□stie` requests *shared* use

[5] The verb `□erase`, however, requires that you identify a file *both* by its tie number and its name, to avoid inadvertent erasure.

of the file; assuming they have the appropriate access permissions, several tasks can share-tie a file at the same time.

Duration of a File Tie

Once you have tied it, a file remains tied until you *erase* it with `⎕erase`, *untie* it with the verb `⎕untie`, or until the end of the task, whichever occurs soonest.

A file tie is *not* affected by changes to the active workspace; in particular, it is unaffected by such actions as clearing the active workspace or loading another saved workspace into the active area. In that regard, a file tie is like a session variable, and unlike a shared variable.

A file tie is not maintained past the lifetime of a task. At the end of a task, SHARP APL automatically unties any files that remain tied, and does not keep any record of what files were tied or what numbers were used to tie them.[6]

Concurrent Use

A task may use many files at the same time, whereas a task has only one active workspace. The maximum number of files that a task may use at one time is a system parameter set by the particular SHARP APL installation; it applies to all tasks. A typical limit is 20 files per task.

Shared Use

Several tasks (including tasks belonging to different users) may have access to the same file at the same time. The file system includes provision for interlocks to permit concurrent updates (that is replacement of the value of an existing component) as well as concurrent read or append.

To manage contention for the same file, or for the same components within a file, among multiple users, the system provides means by which cooperating tasks can signal their desire for temporary exclusive access to all or part of a file.

Library Access Controls

SHARP APL provides the ability to let you control which users, if any, have access to your library and individual files within the library. Access to files is controlled by *file access tables* (See "File Access Controls," below.), and access to libraries is controlled by *library access tables*. Both tables

[6] Thus to be restartable, a task that uses files must establish its own conventions or keep its own records of the files it ties.

have similar structures, but the specific permissions granted within them are different.

The library access table is set by the system verb `□stacl`, described in Chapter 16, "System Nouns and Verbs". When it is set, the library access table defines:

- which user numbers can create and rename files in the library
- which file names are provided as the result of the `□lib` verb
- which user numbers can read and set the library access table.

The library access table is a three-column table, each row of which contains one set of permission information, consisting of a user number, permission code, and passnumber. Each row stipulates three things concerning the use of the file:

User number (column 1) The user number of the owner of a task that may seek access to the library. The number `0` is used to mean "any user."

Permission code (column 2) A number to indicate which capabilities are granted to the user identified in column 1 with the passnumber in column 3.

Passnumber (column 3) An arbitrary integer that the user must supply when accessing the library. Passnumber `0` is equivalent to "no passnumber."

Each individual permission code is a power of 2. Because of that, any combination of permissions may be expressed by the sum of their individual codes. Expressing the permission code in binary shows a `1` for each permitted verb and a `0` for each that is not permitted. The number `¯1` is used to mean "any verb."[7]

Controlled Library Operations

The operations controlled by library access controls, and their associated library access table permissions, are as follows:

Code	*Operation Permitted*
1	Create files in library (`□create`).
2	Rename files into library (`□rename`).
4	List file names in library (using `□lib` from another user). If this code is not specified, no

[7] The binary representation of `¯1` is all `1`s.

file names are listed; if it is specified, only file names that the user has some form of access to are listed.

8 List *all* file names in library (using □*lib* from another user); note that a user must also have permission code 4 to list another user's file names.

16 Read the library access table (□*rdacl*).

32 Set the library access table (□*stacl*).

A typical library access table might look like the following:

```
12345      ¯1       0
67890       3     987
    0      12     654
```

The library access granted by this table is as follows. Note that this access is in addition to the access implicitly associated with each library access table (see below).

- The first row permits user number 12345 unlimited access to the library, with no passnumber.
- The second row permits user number 67890 to create and rename files into the library using passnumber 987.
- The third row permits any user to list all of the file names in the library using passnumber 654, regardless of individual file access controls.

Three rows of default access information are implicitly associated with each library access table. These rows are as follows:

```
library      ¯1        0
      0       4        0
      0       0  (not 0)
```

The first row ensures that a user always has complete access to his own library. This access information takes precedence over the information in the access table set by □*stacl*.

The second and third rows take effect *after* the information in the access table set using □*stacl* and provide default access for users not referred to explicitly in the table.

The second row sets the permission for all users not specified explicitly or implicitly (using 0) in the library access table. This permits users to list the names of only those files in the library that the user number has some

form of access to, unless additional permission is specified elsewhere in the library access table.

The third row specifies no permission (`0`) for all non-zero library passnumbers not explicitly specified using `□stacl`.

File Access Controls

Each file has an *access table*. The access table has three columns. Each row stipulates three things concerning the use of the file:

User number (column 1) The user number of the owner of a task that may seek access to the file. The number `0` is used to mean "any user."

Permission code (column 2) A number to indicate which file verbs may be used by the user identified in column 1 when tied with the passnumber in column 3.

Passnumber (column 3) An arbitrary integer that the user must supply both when tying the file *and* as part of the argument of each verb used with the file (except for three verbs that take no passnumber). Passnumber `0` is equivalent to "no passnumber."

Each individual permission code is a power of 2. Because of that, any combination of permissions may be expressed by the sum of their individual codes. Expressing the permission code in binary shows a `1` for each permitted verb and a `0` for each that is not permitted. The number `¯1` is used to mean "any verb."[8] A table of permission codes appears later in this chapter.

If you are *not* the owner of the file, you may tie it or use verbs with it only as explicitly provided in the file's access table.

The same user number may be mentioned several times in the access table, perhaps with different passnumbers. The *effective permission* is the one corresponding to the *first* occurrence (scanning from the top) of user number paired with the passnumber used to tie the file.

When you *are* the owner of the file, as long as your user number is *not* mentioned in column `0`, you may do anything with the file. It is as though every access table has a hidden last row containing your user number followed by `¯1 0` (meaning "all verbs, no passnumber").

[8] The binary representation of `¯1` is all `1`s.

Where your user number appears in the access table, the rules apply to you in the same way as to anyone else, even though you are the owner. You are restricted to the actions described by the permission code on the first row that contains both your user number and the passnumber with which you tied the file.

Passnumbers and Restrictions on Access

At the time you tie a file, you may be required to supply not one number but two. The first is the arbitrary tie number; what number you use is up to you. The second is a *passnumber*. A passnumber is a positive integer that must match a number stated in the file's access table (described below).

If a file's access table requires a passnumber, you must include the passnumber when you tie the file, and in the argument to almost all verbs you use with the tied file. The passnumber used when you tie the file must be the same one used for all further access to that file, until you untie the file.

The access table may provide more than one way for a user to tie a file. For example, a user might be granted "read only" access when tied with passnumber `12345`, but "read and replace" access when tied with `7863`. See "Permission Codes," below. A task cannot tie the same file in more than one way at one time.

Access Control by User Number and Verb

Every file has an *access table*. The table identifies which users may tie it, what passnumbers (if any) are required of them to do so, and which verbs a user who ties a file with that passnumber may employ with respect to the file.

At its creation, a file has an empty access table. The empty table permits tasks belonging to the file's owner complete access to the file, and denies any access to tasks belonging to anyone else. The owner may change the access table, and thereby define the access that tasks belonging to other users may have to the file. (The access table may even grant to others the right to see or to change the access table.)

Permission Codes

For each combination of user number and passnumber, the access table specifies which verbs a task may use. It is thus possible to set restrictions whose effects are illustrated by the following examples:

- "If a task belonging to user `123456` ties the file with passnumber `1988`, that task may see the size of the file and read any component, but may not do anything else."

- "If a task belonging to user `654321` ties the file with no passnumber, that task may see the file's size, the component information (who wrote it and when) for each component, and may read any component."
- "A task (belonging to any user) that ties the file with passnumber `99` may append new components to the file in a way that does not reveal the file's size, and may change the file's reservation, but may not read any of the components."

There are 17 independent permissions, so a great many other combinations are possible.

The list of permitted verbs is encoded as an integer, called the *permission code*. Each of the separate permissions is a power of 2. The overall permission code that appears in a file's access table is the sum of the codes for the permitted actions. Figure 14–1 shows the values for the separate permissions.

Access Control Values in Base 2	Base 10	Permissions Granted
0000 0000 0000 0000 0001	1	⎕*read*, ⎕*size*
0000 0000 0000 0000 0010	2	⎕*tie*
0000 0000 0000 0000 0100	4	⎕*erase*
0000 0000 0000 0000 1000	8	⎕*append*
0000 0000 0000 0001 0000	16	⎕*replace*
0000 0000 0000 0010 0000	32	⎕*drop*
0000 0000 0000 0100 0000	64	⎕*hold*
0000 0000 0000 1000 0000	128	⎕*rename*
0000 0000 0001 0000 0000	256	⎕*rdac*, ⎕*stac*
0000 0000 0010 0000 0000	512	⎕*rdci*
0000 0000 0100 0000 0000	1024	⎕*resize*
0000 0000 1000 0000 0000	2048	⎕*fhold*
0000 0001 0000 0000 0000	4096	⎕*rdac*
0000 0010 0000 0000 0000	8192	⎕*stac*
0000 0100 0000 0000 0000	16384	⎕*appendr*
0000 1000 0000 0000 0000	32768	⎕*size*
0001 0000 0000 0000 0000	65536	⎕*rdfi*

¯1 Gives unlimited access.

Figure 14–1 : Table of file permission codes.

Tying a File with a Passnumber

As always, the tie number with which you tie a file is up to you; but it must be a number that you are not already using to tie a file. With your tie number, you may include a *passnumber*. If you omit the pass number, it is assumed to be zero. The passnumber must be one of the passnumbers listed for your user in the file's access table. For example, suppose that:

- your user number is `1148629`
- you wish to tie the file *foo* in library `1431816`
- the file's access table looks like this:

```
      0     32768       0
      0       513      99
1148629      1545   38787
2203141      1545   80223
1148629    118527   61216
```

Given those particular entries in the access table:

- Anyone can tie the file with no passnumber (or passnumber `0`). Anyone tying the file with no passnumber has permission `32768`, which permits only `⎕size`.
- None of the permissions includes the ability to see or set the file's access table.
- Anyone can tie the file with passnumber `99`. That permits reading the file and identifying the user that wrote any component.
- You (as user `1148629`) can also tie the file with passnumber `38787` or passnumber `61216`. (However, since you cannot read the file's access table, you must have been given the passnumbers in some other way.[9]
- When you tie the file with passnumber `38787`, you can read any component and you can also append a new component (but not change an existing component). You can also determine the size of the file and of each component, and resize the file. For example, to give the file the tie number `44`, you execute

```
'1431816 foo' ⎕stie 44 38787
```

- When you tie the file with passnumber `61216`, you can do anything to the file (even erase it) except read or change its access table.

9 Note that if you were to write a program that repeatedly tried passnumbers, the system's security facilities would automatically note the fact and take appropriate action.

Having tied the file with a passnumber, you must include the passnumber with the tie number in most uses of the file. The only exceptions are for use of the verbs `⎕untie`, `⎕fhold`, `⎕hold`, and `⎕create`.

Supplying a (non-zero) passnumber when none is required is just as much an error as failing to supply one when it is required. Either action, or using a verb excluded by the permission table, is rejected with the message `file access error`.

Effective Permission

When you tie a file, your effective permission for access to it is controlled by the *first* row in the access table (reading from top to bottom) that matches your user number and passnumber. (Thus if the access table contains several rows that match your user number and passnumber, the later ones have no effect.)

For a file *in your own library*, the access table always contains an implicit last row consisting of your user number followed by `¯1 0`. This implicit last row is effective when:

- you tie the file with no passnumber (or passnumber `0`)

  and

- no earlier row of the access table contains your user number paired with passnumber `0`.

Watch out: It is possible to set a file's access table in such a way that it becomes impossible to do anything useful with it (at least from APL). Of course, all these security measures refer only to access by way of the APL interpreter. The files still exist in the MVS environment, whose requirements are not necessarily those of APL.

The expression `(20⍴2)⊤`***perm*** returns a 20-item Boolean list containing a `1` for each permission granted and a `0` for those denied. Using it to compress rows of a table such as the one shown in Figure 14–1 returns a list of the permitted verbs.

Concurrency

SHARP APL provides a way for two or more tasks running at the same time to coordinate the use of files they share. When several tasks work on the same files at the same time, certain of their activities (but by no means all) have to be isolated. Certain activities are sensitive. Once you have started such an activity, you need to complete it without the risk that one of the other tasks will do something that interferes.

Suppose you have a program that reads a table from a file, alters some of the entries, and then puts it back. If two tasks do that at the same time, one update may overwrite the other; the update written later may be ignorant of the one completed just before it. So you need assurance that no one else tries to perform the same sort of activity at the same time. You need to isolate each sensitive activity so that:

- You do not start a sensitive activity while someone else is engaged in something that might conflict.
- Once you do start a sensitive activity, no one else can start a potentially conflicting activity until you have finished.

The file-holding verbs `⎕fhold` and `⎕hold` set up such protections. However, what you get is not a guarantee but rather a set of tools to facilitate voluntary restraint. The *hold* verbs give you a way to make your task wait until other sensitive work is done, and a way to signal cooperating tasks that you need exclusive use of the files. Other tasks will respect your request for exclusive use *if* they also use the hold verbs appropriately. The hold verbs do *not* assure that you have the files to yourself; only the exclusive tie verb `⎕tie` can do that. It is your responsibility as application designer to be sure that you have identified the sensitive parts of your procedures, and isolated each of them with appropriate use of the hold verbs.

The two forms of the verb (`⎕hold` and `⎕fhold`) have the same effect, and operate on a common queue of interlocked file events. They differ only in the forms of their arguments. The older form, `⎕hold`, has no provision for passnumbers and may not be used dyadically.

The right argument ω is an integer array containing distinct tie numbers. For `⎕fhold`, ω may also be a multi-row table; when the argument is a table, the first row contains tie numbers and the second row the associated passnumbers required for reference to the file (or `0` for each file tied without a passnumber). Because `⎕hold` is unaffected by passnumbers, a file used for an application requiring passnumbers is usually given an access table prohibiting use of `⎕hold`.

The third and fourth rows, if specified, are the lower and upper component numbers affected by the hold. This allows holding part of a file, while

leaving the remainder of it free for use by others. It also permits creation of general synchronization mechanisms, since the component numbers in the range need not exist. See the description of `□fhold` in Chapter 16, "System Nouns and Verbs".

If you did not specify passnumbers when you tied the files, the argument may be a list or (if you are holding only one file) an item; `□fhold` also accepts a one-row table.

`1 □fhold ⍵` can specify files which you already hold as long as the range specified exactly matches the range already held.

`□hold` or `□fhold` of a file without specifying a range results in a default range of `¯2147483648` to `2147483647`.

Monadic Use of the Hold Verbs

Monadic use of either of the hold verbs has the following effects:

1. All preceding holds (if any) are released. It does not matter whether the new request to hold applies to the same files or to different ones. Each execution of either of the hold verbs cancels any preceding use. (For more controlled release, see dyadic use, below.)
2. Execution of your task goes into an indefinite delay until all the files you have asked to hold are free of holds set by other tasks.

   From the keyboard, you can signal an interrupt. If the interrupt is not trapped, that returns control to the keyboard and cancels the request to hold. But a successfully trapped interrupt does not break the hold.
3. Once all the files you have requested are free, your hold takes effect. Execution of your task proceeds.

When the interpreter receives from another task a request to hold any of the files you are holding, that task goes into an indefinite delay until either you release your hold on the files the other task has asked for, or you untie them.

Dyadic Use of the Hold Verbs

When `□fhold` is used dyadically, the right argument is a list of tie numbers, or a table of tie numbers, passnumbers, and hold ranges in the same form as described for monadic use. The left argument is an integer item whose value is either `1` or `¯1`. The effect of the left argument is as follows:

`1` ***Selective hold***: The files identified in ⍵ are added to the hold queue without releasing any files previously held.

`¯1` ***Selective release***: The files identified in ⍵ are released without releasing any files already held but not mentioned in ⍵.

Ending a File Hold

A file hold, once in effect, is ended by any of the following:

- The appropriate dyadic use of `⎕fhold` (see below).
- Any monadic use of `⎕hold` or `⎕fhold`.
- Any untrapped error or interrupt.
- Untying the affected file or files. (When you untie some but not all of the files you are holding, the hold remains in effect for those that are still tied.)
- The end of your task. A freestanding S-task is ended when it reaches a normal end of execution, or when it encounters an error. A T-task is ended when you sign off, or as a consequence of a communications failure.
- Any return to immediate execution (including one trapped as event 2001, since the only actions possible with event 2001 lead ultimately to immediate execution).

  You might return to immediate execution because the defined verb that issued the hold has come to a normal end. Or you might return to immediate execution because the system encounters an error, or because you signaled an interrupt.

Watch out: You cannot experiment with file holds by entering statements one after another in immediate-execution mode. Any hold you enter that way terminates immediately when the system returns for the next entry from the keyboard. In that situation, the most you could do would be to put into a single line of entry (separated by diamonds) both the file hold and the statement you want executed during the hold.

When the system returns to the keyboard to receive ⎕- or ⍞-input, that does *not* constitute a return to immediate-execution mode, and does *not* terminate a hold.

Normal Use of File Hold

File holds isolate a segment of sensitive code, known as a *critical section.* They therefore occur inside the definition of a defined verb, in a sequence something like this:

```
    ∇ Verb ...
[1]   ...
[2]   ...
[3]   1 ⎕fhold ⍵
[4]   Critical Section
[5]   ¯1 ⎕fhold ⍵
[6]   ...
```

It is good programming practice to minimize the elapsed time during which a file hold will be in effect. Compare the following two examples. In the first, the request for input occurs inside the hold, whereas in the second, the hold does not occur until the request for input has been satisfied.

Not good:

```
f← 1 ⎕fhold ⍵
x←((⎕read ⍵,⍺), ⎕fi⍞) ⎕replace ⍵,⍺
f←¯1 ⎕fhold ⍵
```

Better:

```
t←⎕fi ⍞
f← 1 ⎕fhold ⍵
x←((⎕read ⍵,⍺),t) ⎕replace ⍵,⍺
f←¯1 ⎕fhold ⍵
```

Watch out: Inside the block spanned by your initial hold and the subsequent release, your *critical section* must not invoke a verb that uses a monadic file hold. If you did that, the verb you called would undo your earlier file hold when it issued its own. For this reason, a utility verb should not execute monadic file holds.

Summary of File Verbs

The following section groups the various file verbs according to their function. Their full definitions are in the alphabetical list of system nouns and verbs in Chapter 16, "System Nouns and Verbs".

Abbreviations: Where the argument of a file verb must have a specific form or content, rather than the symbols ⍺ or ⍵ for its arguments, the list that follows uses the following abbreviations:

A ***File access table.*** A three-column table describing permitted access to a file.
Column 1: User number.
Column 2: Passnumber.
Column 3: Permission code.

L ***Library access table.*** A three-column table describing permitted access to a library.
Column 1: User number.
Column 2: Passnumber.
Column 3: Permission code.

u ***User or library number.*** The owner of a task seeking access to a library or file.

perm ***Permission code.*** A numeric encoding of the actions permitted to a particular combination of user number and passnumber.

p ***File passnumber.*** An arbitrary number which a task using a library or file must supply when tying or referencing a file.

lp ***Library passnumber.*** An arbitrary number which a task using a library or file must supply when referencing a library.

c ***Component number.*** An integer identifying one of the components of a file.

t ***Tie number.*** An arbitrary number (selected by the task that uses a file) or list of numbers by which the task refers to the file, or a collection of files.

fn ***File name.*** A character list containing the file's name. When the owner of the file differs from the owner of the task, the name must include the user number of the file's owner, as characters, separated from the rest of the name by a blank. For example, `'123 data'`. A few file verbs (`⎕create` and `⎕rename`) also allow an optional file size to be specified after the file name, as part of this argument.

File Verbs: Inquiry

`⎕avail ⍳0`	**File system availability.**
`⎕lib` ***u*** `[,`***lpn***`]`	**Library** of files in library ***u*** to which you have any sort of access, with optional library passnumber ***lpn***.
`⎕names`	**Names** of currently tied files
`⎕nums`	**Tie numbers** of currently tied files.

File Verbs: Existence

fn `⎕create` ***t*** `[`***c*** `[,`***lpn***`]]`	**Create** a file with name ***fn***, share-tied to ***t***, with optional starting component number ***c***, and optional library passnumber ***lpn***.
fn `⎕rename` ***t*** `[,`***pn*** `[,`***lpn***`]]`	**Rename** the file tied to ***t*** so its name becomes ***fn***, with optional file passnumber ***pn*** and optional library passnumber ***lpn***.
fn `⎕erase` ***t***`[,`***pn***`]`	**Erase** the file tied to ***t*** with optional passnumber ***pn*** whose name is ***fn***.

File Verbs: Linkage

fn `⎕stie` ***t***`[,`***pn***`]`	**Share-tie** to number ***t*** the file named ***fn***, with optional passnumber ***pn***.
fn `⎕tie` ***t***`[,`***pn***`]`	**Tie** to number ***t*** the file named ***fn***, for exclusive use by this task, with optional pass number ***pn***.
`⎕untie` ***t***	**Untie** the files tied to the list of tie numbers ***t***.

File Verbs: Description and Limits

`⎕rdci` ***t, c*** `[,`***pn***`]`	**Read component information** for component ***c*** of the file tied to number ***t*** with optional passnumber ***pn***.
`⎕rdfi` ***t*** `[,`***pn***`]`	**Read file information** for the file tied to number ***t*** with optional passnumber ***pn***.
`⎕size` ***t*** `[,`***pn***`]`	**Size** of the file tied to ***t***, with optional passnumber ***pn***.
⍺ `⎕resize` ***t***`[,`***pn***`]`	**Resize** to ⍺ the size limit of the file tied to ***t***, with optional passnumber ***pn***.

File Verbs: File and Library Access

`⎕rdac` ***t*** `[,`***pn***`]`	**Read file access table** of file tied to ***t***, with optional passnumber ***pn***.
`⎕rdacl` ***u*** `[,`***lpn***`]`	**Read library access table** of library ***u***, using optional library passnumber ***lpn***.
A `⎕stac` ***t*** `[,`***pn***`]`	**Set access table** to ***A*** for the file tied to ***t***, with optional passnumber ***pn***.
L `⎕stacl` ***u*** `[,`***lpn***`]`	**Set library access table** to ***L*** for library ***u***, using optional passnumber ***lpn***.

File Verbs: Concurrency

`⎕hold` ***t***	**File hold.** Delay `⎕hold` or `⎕fhold` by other tasks to the files tied to ***t***, until the next `⎕hold`, `⎕fhold`, or return to immediate execution. All prior holds are released.
`⎕fhold` ⍵	**File hold.** Same as `⎕hold`, but optionally ⍵ may be a table including hold ranges.
⍺ `⎕fhold` ⍵	**File hold.** Same as monad `⎕fhold`, except that other held files are unaffected, and holds may be added or removed.

File Verbs: Use

`⎕read` ***t, c*** `[,`***pn***`]`	**Read** data object in component ***c*** of file tied to ***t***, with optional passnumber ***pn***.
⍺ `⎕replace` ***t, c*** `[,`***pn***`]`	**Replace** component ***c*** of file tied to ***t***, with optional passnumber ***pn***, with noun ⍺.
⍺ `⎕appendr` ***t*** `[,`***pn***`]`	**Append with result.** Write new component containing noun ⍺ in file tied to ***t***, with optional passnumber ***pn***, providing new component number as result.
⍺ `⎕append` ***t*** `[,`***pn***`]`	**Append.** Effect same as `⎕appendr`, but no result.
`⎕drop` ***t, n*** `[,`***pn***`]`	**Drop** `|`***n*** components from file tied to ***t***, with optional passnumber ***pn***.

15 *APL Characters*

This chapter concerns the characters used to write APL sentences and the means to enter and display them. Transactions between keyboard and computer may involve a series of translations, as may transactions between computer and display. To help distinguish some separate issues, we start by reviewing some terms and concepts.

Distinctions Between Characters and Their Representations

- ***Glyphs.*** The set of shapes used to write sentences in the APL language.
- ***Internal characters.*** The set of binary patterns used within the SHARP APL system to represent character data in general, and in particular to represent the characters used in APL sentences.
- ***Keystrokes.*** The buttons, or sequence of buttons, that the user at a keyboard presses to evoke the various characters.
- ***Transmissions.*** The codes used to transmit information about characters or keystrokes between the terminal and the central system.

  In a "dumb terminal," the transmission may describe the sequence of keystrokes, leaving to the central system the task of interpreting them as characters. In a "smart terminal" emulator, the terminal may provide local processing to analyze the keystrokes and then transmit codes that represent the target character rather than the keystrokes themselves.

- ***Display font.*** Associated with a keyboard is a display screen or printer. The display shows characters that represent the input from the keyboard and the output from the computer. The shape of each displayed character is determined by a font stored in the device. During input or output, glyphs from that font are selected and displayed.

- ***Local vs. echoed input.*** During input from the keyboard, the terminal may use either of two systems to generate characters on the screen:
  - *Echoed display.* The keyboard causes a transmission to the central computer. The central computer (or, in a network, an intermediate computer) transmits back to the terminal the instructions on what to display. The computer is said to "echo" what it has received. This system is sometimes called "full duplex."

- *Local display.* The keyboard causes both a transmission to the central computer *and* the display at the terminal. The terminal is said to use a "local echo" of what was sent. This system is sometimes mistakenly called "half duplex" (although in APL systems the transmission line between terminal and host is always full duplex).

Mappings Between Keys, Glyphs, and Characters

In principle, there is a simple correspondence between characters in the computer, glyphs on screen or paper, and keys on the keyboard.

In practice, complications arise from two sources:

- ***Systems of character encoding.*** Historically, countries and manufacturers have encoded characters differently. In the ASCII and ISO standards, the letter A is encoded as 65, but in the EBCDIC system used in mainframes made by IBM and several other manufacturers, the letter A is encoded as 113. These encoding systems are independent and self-contained. Translation between them is necessary when data or devices are mixed: for example, when an APL system using EBCDIC processes a tape written in ASCII, or accepts a terminal that uses ASCII transmission codes.
- ***National extensions to the character set.*** English uses only 26 distinct characters, with no diacritical marks. Other European languages use varying numbers of additional marks or characters. In general, each country uses a set of character codes sufficient to its own needs, but without regard to characters in other languages.[1]
- ***APL extensions to the character set.*** APL makes use of symbols to represent its primitives. Internally, the APL system must assign codes from its set of recognized characters to represent those characters. Similarly, the devices used for input and output must accommodate APL characters in their keyboards, their display fonts, and their transmission systems. The various schemes for doing this have not always been consistent with other uses of the device, or with each other.

Characters in the APL System

The SHARP APL system is based on 8-bit characters, permitting 256 distinct encodings. In a character array, an item may be any of those characters.

The SHARP APL character set comprises the following components:

[1] Indeed, even restricting the discussion to writing systems based on the Latin alphabet, providing a unique encoding for APL and for every combination of letter and diacritical mark used somewhere in Europe, would exhaust not only the 128 characters representable in 7 bits, but also the 256 characters representable in 8 bits.

Basic Alphabet

Fifty-two characters, including two distinct forms of the 26 letters of the English alphabet:

a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

In addition, the characters ∆ and ⍙ are also permitted in names; in that sense, they are part of the basic alphabet.

Numerals

The 10 characters 0 1 2 3 4 5 6 7 8 9.

Symbols

The following 80 symbols are recognized as members of the APL character set (even though a few have no assigned use):

¨ ¯ < ≤ = ≡ ≥ > ≠ ∨ ⍱ ∧ ⍲ + − × ÷ ⌸ ◇ ⊆
⍰ ⍵ ∈ ⍴ ∼ ↑ ↓ ⍳ ○ ⍜ ⌽ ⊖ ⍉ ⍟ ⋆ → ← ⊢ ⊣ ⍸
⍺ ⌈ ⌊ _ ∇ ⍛ ⍒ ∆ ⍋ ∘ ⍤ ⍎ ' ⍁ ⍂ [ ] ( ) {
⊂ ⊃ ∩ ∪ ⊤ ⍑ ⊥ ⍕ ⌶ | , ⍪ ; . : / ⌿ \ ⍀ }

Graphics

The following 11 box-drawing characters may be included in a character array or in comments but not as names or symbols. They are displayable or enterable only on certain devices.

| ┤ ├ ┴ ┬ ─ ┼ ┘ └ ┌ ┐

The class *graphic characters* also includes the 26 letters of the "third alphabet" (discussed below).

National Use Characters

Fourteen character codes have been reserved for glyphs that occur in some European languages but not others. The glyphs associated with these *national use* characters vary, depending on the conventions of the country in which they are used.[2] Like the graphics characters, these may be displayed but may not be used in the names of APL nouns or verbs.

[2] Thus the point of the term *national use* is that these characters are inappropriate for *international* use, since the same character is given different keys or displays in different countries.

To provide an unambiguous way of talking about them, these characters are assigned numbers from 1 to 14. The table below shows the appearance of each in Germany and in North America.

NU number	1	2	3	4	5	6	7	8	9	10	11	12	13	14
German	Ä	Ü	ö	`	$	#	§	^	ß	ä	ü	Ö	!	"
North-American English	¢	|	!	$	¬	¦	`	#	@	"	~	{	}	\

Control Characters

The SHARP APL system recognizes *newline* as a delimiter between lines of a definition fixed with 3 ⎕`fd`. The system's session handlers recognize the characters *null, idle, backspace, newline, and line feed.*[3] The SHARP APL system does not recognize *tab* characters as such. However, the system noun ⎕`ht` allows session handlers to properly interpret tab characters transmitted to the APL system, so that they appear as an appropriate number of blanks when they reach the APL workspace. Similarly on output, session handlers replace sequences of blanks with tab characters with tab characters, to speed up transmission.

Normal and Reversed Alphabets

APL is in a period of transition between two conventions regarding the way alphabetic characters are written. Both the old convention (called *reverse*) and the new (called *normal*) offer two ways of writing each of the 26 letters. The normal and reversed styles are as follows:

Normal style: `abcdefghijklmnopqrstuvwxyz`
`ABCDEFGHIJKLMNOPQRSTUVWXYZ`

Reversed style: `ABCDEFGHIJKLMNOPQRSTUVWXYZ`
`abcdefghijklmnopqrstuvwxyz`

For work from terminals or terminal emulators that require a style opposite to that assumed by SHARP APL or by the application, the system command `)cc` (for *character control*) permits you to select one or the other. The command `)cc normal` or `)cc reverse` instructs the session handler to

[3] The action here called *newline* means that the cursor is to be moved to the beginning of the next line down; ASCII devices treat this as two actions: (1) returning the cursor to the beginning of the current line, and (2) moving down one line. When the system communicates with ASCII devices, the device drivers convert the single *newline* character to the sequence *carriage return, line feed* required by the device. The action here called *line feed* moves the cursor down without changing its horizontal position.

translate accordingly.[4] The defined verbs `cc` and `Cc`, available in public workspace `1 wsfns`, provide the same capability under program control.

Historical Perspective on the Alphabet Change

The reversed style was invented first. It accommodated printing terminals whose total number of distinct characters was severely restricted by the number of positions on the metal type element used to print the characters. It had room for only 88 shapes. (Some later devices increased this to 96.) In order to provide positions for glyphs from which to construct the APL characters, two economies were adopted:

- ***Overstriking to build APL characters.*** Many of the APL characters are constructed from basic shapes that can be combined. This not only gives APL an open-ended character set, but permits a major economy of space. For example, ÷ and ⍞ can be combined to form ⍰. The type element need contain glyphs only for ÷ and ⍞ because it could form the glyph for the character ⍰ by overstriking the other two.[5]
- ***Overstriking to differentiate case.*** In most European languages, each letter exists in two shapes, called majuscule and minuscule (or capital and lowercase). To print both forms of the 26 letters of English requires 52 glyphs. The economy adopted in early APL systems was to dispense with the minuscules (freeing space on the type element for APL symbols) and using overstrike to differentiate the two forms of each letter. Instead of `a` and `A`, they wrote `A` and `A̲`, which could be formed by overstriking `A` with underbar. Thus, by overstriking each of the 26 letters with one additional character, 26 alternate forms could be constructed.

Now that there are devices that permit the intermixing of APL characters with both lowercase and capital letters, it is no longer necessary to confine APL's ordinary alphabet to capitals. The normal style permits a more natural use of letters in APL names and text, and reflects the typing habits of people who are accustomed to using common text entry devices such as typewriters and personal computers.

Text Support for Non-APL Terminals

While it is important to be able to show APL characters at a device that will read or write APL programs, many applications require APL characters neither in input nor in display. Such an application can be run from a device

[4] The options also differ in their interpretation of backspace from non-APL terminals, and in the way they treat characters that cannot be directly entered or displayed; see the description of `)cc` in Chapter 17, "System Commands".

[5] Note that ⍰ is just one character. However, in order to print it, the output drivers had to transmit ⍞ *backspace* ÷. For the same reason, the input handlers had to be able to recognize that a sequence that placed ⍞ and ÷ at the same physical position denoted the single character ⍰.

that has support neither for APL characters nor for overstrikes. In this situation, different manufacturers have made opposite assumptions about the mapping between the two forms of the alphabet in APL and the two forms displayed on the device. Because of this inconsistency, what one device sees as

```
Some Of These Letters Are Capitalized
```

may appear to the other as

```
sOME oF tHESE lETTERS aRE cAPITALIZED
```

The command `)cc` permits you to compensate for these opposite conventions.

With the setting `)cc normal`, to non-APL devices the primary alphabet is transmitted as lowercase, and the secondary alphabet as capitals. That is the way all examples in this manual are shown.

With the setting `)cc reverse`, the situation is reversed. That is, to non-APL devices, the primary alphabet is transmitted as capitals, and the secondary as lowercase. Thus, while it is preferable to operate consistently with the normal convention, where need be you can preserve the intent of a particular text by switching to reverse. See the description of the `)cc` command in Chapter 17, "System Commands".

The Third Alphabet

When it first became possible to intermix APL with both majuscule and minuscule letters, some devices introduced the minuscules (for example, a) while continuing to support both A and A. This unfortunately gave APL not two forms of each letter, but three. The existence of some devices with three alphabets and others with only two has led to confusion about how to convert between them.

SHARP APL continues to accept input from devices that generate third-alphabet input. However, third alphabet characters have at no time been permitted in the names of APL objects or the spelling of system commands, and are not permitted now.

Keyboard Alternatives

The original APL keyboard (designed for the IBM 1050 and 2741 terminals) assumed that the APL character set *replaced* whatever characters the machine might otherwise have used. The keyboard assigned glyphs to keys by standards of internal consistency, not necessarily consistency with other uses of the device.

The next generation of display and dot matrix printing devices were switchable: the same machine could act either as a standard terminal (with its keyboard and characters) or an APL device (with its keyboard and characters). The machine (and hence the person using the machine) operated in either of two modes: a standard EBCDIC or ASCII mode, and an APL mode. This solution embodied in electronics the solution imposed by the fact that on the old machines you changed modes by physically replacing the type element. As an unfortunate consequence, the same keystroke that meant ⍺ when sent to an APL device, meant `A` when sent to a non-APL device. Similarly, a data array that represented text was ambiguous: the same internal character produced ⍺ at an APL device but `A` at a non-APL device.

Modern devices instead assume that APL characters *augment* the standard keys and characters, rather than replace them. In devices that have a *union*[6] character set, glyphs that are common to both APL and the standard (non-APL) character set are keyed in the standard way. The extra APL symbols are provided by some sort of extra keying (the *Alt* key, overstrike sequences, and so on).

Maintaining Data Independence

In the design of applications, it is important to avoid the creation of dependencies on external devices. It is strongly recommended that:

- In all stored data, the *same* encoding of alphabetic characters be employed throughout

  and

- The universal encoding be employed, to wit:
  - *first* alphabet to represent minuscules
  - *second* alphabet to represent majuscules
  - *third* alphabet not at all in the internal representation of data.

- Where the application must support devices that require different encodings, the translation to alternative characters be located in the session handler, if possible. Otherwise, the translation should be provided by the application's input-output handlers, so that the internal data remains unaffected.

Internal Representation of APL Characters

Each of the 256 possible characters is represented internally by a pattern of 8 bits. The system noun `⎕av` consists of all 256 characters, arranged in

[6] So-called because it supports the union of the two sets of characters.

the order by the numeric value of their representations. For example, with index-origin 0,

```
      (8⍴2)⊤⎕←⎕av⍳'÷'
29
0 0 0 1 1 1 0 1
```

demonstrates that the internal representation of the divide character ÷ is `00011101`, whose numeric value is 29.

For most purposes, you do not need to know how characters are represented internally. However, the information is useful when you need to interpret data imported from outside the APL context, or to produce data for export to a non-APL context. It may also be useful to generate characters that cannot be entered from the keyboard of the device you are using.

The codes used to represent APL characters are arbitrary and differ from one APL system to another. For material transmitted to or from a different APL system, you should consult its documentation for details of the way it represents its characters. The assignment of characters to internal codes for the SHARP APL system is shown in Figure 15–1.

Translation for the Import and Export of Data

Character data imported from or exported to a file external to the APL system may require translation in order to be readable. In general, such a translation is obtained by an expression such as

```
trans[⎕av⍳x]
```

where *x* is the array to be translated and *trans* is a translate table.

You get the translation in the reverse direction by:

```
⎕av[trans⍳x]
```

To assure that such a translation can handle any argument and that the translation is completely reversible, it is essential to make *trans* a *permutation* of ⎕av. That is, *trans* should contain every member of ⎕av exactly once.

Utilities provided on the host to read and write external files may include their own translate tables, and offer options to perform translation automatically.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	†	†			n4	n1						[	]	†		
16	(	)	;	/	\	←	→			⋄	+	−	×	÷	*	⌈
32	⌊	|	∧	∨	<	≤	=	≥	>	≠	∝	∈	⍳	⍴	⍵	,
48	!	⌽	⊥	⊤	○	?	~	↑	↓	⊂	⊃	∩	∪	_	⍉	⌶
64	∘	⍞	⍠	⍟	⍲	⍱	⍝	⍋	⍒	⍬	⍲	⍱	⍇	⍐	⍋	◇
80	⊣	⊢	}	{			a	b	c	d	e	f	g	h	i	j
96	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
112	Δ	A	B	C	D	E	F	G	H	I	J	K	K	M	N	O
128	P	Q	R	S	T	U	V	W	X	Y	Z	⍙	0	1	2	3
144	4	5	6	7	8	9	.	¯	†	'	:	∇	†		†	†
160	⍢			≡	⍤	Ö	a	b	c	d	e	f	g	h	i	j
176	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
192		n2	n3		n5	n6	n7	n8	n9	n10	n11	n12	n13	n14	%	&
208		⍳	∈									⍨				
224																
240		|	┤	├	┴	┬	─	┼	┘	└	┌	┐	†	†	†	

† *Control characters*
0 Idle
1 Null
152 Blank
156 Newline; CR
158 Backspace
159 Line feed
252 Full-screen field mark
253 Full-screen DUP mark
254 Full-screen null
n-- National use character

Figure 15–1 : Internal representation of APL characters.

Transmission Codes and Overstrike Conventions

Key to Columns

1 ***Index*** (zero-origin) in `⎕av`.

2 ***APL character.*** Appearance of *national use characters* (named NU1, NU2, etc.) varies by country. The characters shown are those for North-American English.

3 ***Overstrike pairs*** to construct the character during input, by overstriking, or with an overstrike escape sequence; the order within a pair is not significant.

4 ***Description.*** Where a character has an established name (for example, *dieresis*, *rho*, etc.) that name is shown; however, some characters have no agreed-on name, and so are described by the functions that they denote.

5 ***Non-APL.*** Character to enter from an ASCII or Teletype keyboard not equipped with APL and signed on in "native" mode (using right parenthesis above 9 or 0).

6 ***APL from a non-APL device.*** Character to enter from an ASCII keyboard without APL keytops, but signed on as if for APL (right parenthesis two keys to the right of the letter L). Where two characters are shown, the first is for typewriter-paired ASCII, and the second is for bit-paired ASCII.

7 ***Telex.*** Character to enter from a Telex keyboard. FS means "figure shift."

8 ***APL character.*** Repeated for ease of reading.

1	2	3	4	5	6	7	8
0			Idle: If sent to display, no effect				
1			NUL: Not sent				
2 – 3			unused				
4	$	S|	NU 4	$	~ ¦	FSH	$
5	¢	c|	NU 1				¢
6 – 13			unused				
14	[		Left bracket	[	;		[
15	]		Right bracket	]	' :		]

1	2	3	4	5	6	7	8
16	(		Left parenthesis	(	: +	(	(
17	)		Right parenthesis	)	" *	)	)
18	;		Semicolon	;	<		;
19	/		Slash	/	/		/
20	\		Backslash	\	?		\
21	←		Left arrow	–	@	=	←
22	→		Right arrow	@	] `	/	→
23 – 24			unused				
25	¨		Dieresis		!		¨
26	+		Plus	+	–	+	+
27	–		Minus	–	_ =	FSF	–
28	×		Times	#	= ^		×
29	÷		Divide	%	+ ~		÷
30	*		Star	*	P		*
31	⌈		Max		S		⌈
32	⌊		Min		D		⌊
33	|		Stile	¦	M		|
34	^		Caret	^	) –		^
35	v		Inverted caret		( )		v
36	<		Less	<	#		<
37	≤		Not greater		$		≤
38	=		Equals	=	%		=
39	≥		Not less		^ &		≥
40	>		Greater	>	& '		>
41	≠	=/	Not equal		* (		≠
42	⍺		Alpha		A		⍺
43	∈		Epsilon		E		∈
44	⍳		Iota		I		⍳
45	⍴		Rho	&	R		⍴
46	⍵		Omega		W		⍵
47	,		Comma	,	,	,	,
48	!	'.	Shriek	!			!

1	2	3	4	5	6	7	8
49	⌽	○|	Rotate				⌽
50	⊥		Base value		*B*		⊥
51	⊤		Represent		*N*		⊤
52	○		Circle		*O*		○
53	*?*		Query	*?*	*Q*	*?*	*?*
54	~		Tilde	~	*T*		~
55	↑		Up arrow	*Y*		↑	
56	↓		Down arrow	*U*		↓	
57	⊂		Shoe		*Z*		⊂
58	⊃		Link		*X*		⊃
59	∩		Cap		*C*		∩
60	∪		Cup		*V*		∪
61	_		Underbar		*F*		_
62	⍉	○\	Transpose				⍉
63	⌶	⊤⊥	I-beam				⌶
64	∘		Jot		*J*		∘
65	□		Quad	`	*L*	FSG	□
66	⍞	□'	Quote-quad				⍞
67	⍟	○*	Log				⍟
68	⍲	∧~	Nand				⍲
69	⍱	∨~	Nor				⍱
70	⍝	∩∘	Lamp				⍝
71	⍋	Δ|	Grade up				⍋
72	⍒	∇|	Grade down				⍒
73	⊖	○–	Rotate first				⊖
74	⌿	/–	Over first				⌿
75	⍀	\–	Scan first				⍀
76	⌹	□÷	Domino				⌹
77	⍤	⊤∘	Thorn				⍤
78	⍎	⊥∘	Execute				⍎
79	◇	<>	Diamond		` \		◇
80	⊣	–)	Lev		¦ }		⊣

1	2	3	4	5	6	7	8
81	⊢	(−	Dex		\ [		⊢
82	}	⍥]	Right brace	}	}		}
83	{	[⍥	Left brace	{	{]		{
84 – 85			unused				
86 – 111	a–z		First alphabet				a–z
112	Δ		Delta		H		Δ
113 – 138	A–Z	a–z _	Second alphabet				A–Z
139	⍙	Δ_	Delta underbar				⍙
140 – 149	0–9		Digits zero through nine				0–9
150	.		Dot	.	.	.	.
151	¯		Macron	"	@ "	–	¯
152			BLANK				
153	'		Quote	'	K		'
154	:		Colon	:	>	:	:
155	∇		Del		G		∇
156			Newline; CR				
157			unused			'	
158			Backspace				
159			Line feed				
160	⍢	∇~	Del tilde				⍢
161			unused				
162			unused				
163	≡	=_	Match				≡
164	⍤	⍥⍤	Paw				⍤
165	⍜	○⍤	Hoof				⍜
166 – 191	a–z		Third alphabet				a–z
192			unused				
193	|	|⊥	NU 2				|
194	!	'⍥	NU3				!
195			unused				
196	¬	~/	NU 5				¬
197	¦	',	NU 6				¦

1	2	3	4	5	6	7	1
198	`	\¯	NU 7				`
199	#	*N*=	NU 8				#
200	@	*Q*∘	NU 9				@
201	"	¨,	NU 10				"
202	~	~¯	NU 11				~
203	{	[○	NU 12				{
204	}	○]	NU 13				}
205	\	|	NU 14				\
206	%	÷/	Percent				%
207	&	α\	Ampersand				&
208			unused				
209	⍳	⍳_	Iota bar				⍳
210	∈	∈_	Epsilon bar				∈
211 – 218			unused				
219	⍪	−,	Comma bar				⍪
220 – 240			unused				
241 – 251			Box-drawing characters (see below)				
252			Field mark				
253			DUP mark				
254			Full-screen null				
255			unused				

Box-drawing characters are displayed (on devices that support them) by selecting from ⎕*av* at the positions shown to the right of the box, below.

250	246	245	246	251
241		241		241
243	246	247	246	242
241		241		241
249	246	244	246	248

16 *System Nouns and Verbs*

System verbs are primitives without symbols. They are primitive in the sense that they are "built in," and need no definition. However, unlike the main body of primitives, they have not been assigned symbols. Instead, each has a fixed *distinguished name* that starts with the symbol ⎕ (*quad*) or ⍞ (*quote-quad*), possibly followed by some letters. For example, ⎕`cr` returns the *canonical representation* of a user-defined verb; ⎕`svo` offers to share a name with another processor. Since you are not permitted to use ⎕ and ⍞ in user-assigned names, these distinguished names are distinct from any names you could give to user-defined objects.

System nouns are names that are always shared with the APL system. You can refer to their values, but so can the system. Like system verbs, the systems nouns have fixed distinguished names. The system attaches special significance to each. For example, ⎕`io` establishes the *index origin* (which may be 1 or 0), and ⎕`pw` establishes the *page width* for displays.

Certain system names report the current value of system statistics. For example, ⎕`ts` contains the *timestamp*, reporting the date and time, and ⎕`ai` contains *accounting information* regarding the current user. For most purposes, it is moot whether these names denote nouns, or verbs that take no argument. This manual describes them all as nouns. Figure 16–1 indicates which you can set.[1]

[1] This is a point about which APL systems differ. In SHARP APL, the names whose values cannot be set are implemented as if they were niladic verbs; an attempt to set one is rejected as a `syntax error`.

System Noun	*You Set*	
`⎕ai`	—	Accounting information
`⎕av`	—	Atomic vector
`⎕ct`	Yes	Comparison tolerance
`⎕ec`	Yes	Environment condition
`⎕er`	Yes	Event report
`⎕fc`	Yes	Format control
`⎕ht`	Yes	Horizontal tab
`⎕io`	Yes	Index origin
`⎕lx`	Yes	Latent expression
`⎕lc`	—	Line counter
`⎕names`	—	Names of tied files
`⎕nums`	—	Numbers of tied files
`⎕pp`	Yes	Printing precision
`⎕ps`	Yes	Position and spacing
`⎕pw`	Yes	Page width
`⎕rl`	Yes	Random link
`⎕runs`	—	Running tasks
`⎕sc`	Yes	State change
`⎕ts`	—	Timestamp
`⎕trap`	Yes	Trap expressions
`⎕ul`	—	User load
`⎕wa`	—	Work area

Figure 16–1 : System nouns, indicating which you can set.

Results of System Verbs

A few of the system verbs return no result (and do their work as side effects). This is noted by *no result* in the descriptions that follow.

Alphabetical List of System Names

What follows is an alphabetically arranged summary that lists all the system names. Each entry presents a description, and may also point to related definitions or to discussion in other chapters.

⎕ai

Accounting Information

System noun ⎕ai provides information about resource usage by your user number.

Result: A nine-element numeric list containing the following:

⎕ai[1] The user number of the task's owner.

⎕ai[2] CPU time used by this APL task, cumulative since start of the task, in milliunits. A unit is defined as a second of actual CPU time multiplied by a performance factor intended to give comparable results for different CPUs. However, since the "speed" of a machine does not affect all operations in the same way, CPU figures obtained on different machines are only roughly comparable.

⎕ai[3] Elapsed time since the start of this APL task, in milliseconds.

⎕ai[4] Keying time, cumulative since the start of this APL task, in milliseconds. Keying time is the interval during which the system is prepared to accept input from the keyboard, in any mode. Hence this is the time during which you *could* have been keying, whether or not you were.

⎕ai[5] Characters transmitted, cumulative since the start of this APL task. This includes all input and output exchanged with a terminal, or with the controlling shared name of an S-task.

⎕ai[6] Like ⎕ai[2], but cumulative for all T- and S-tasks on this user number since the last billing.

⎕ai[7] Like ⎕ai[3], but cumulative for all T- and S-tasks on this user number since the last billing.

⎕ai[8] Zero (unused).

⎕ai[9] Like ⎕ai[5], but cumulative for all T- and S-tasks on this user number since the last billing.

□*append*

Append Noun to Tied File

Noun □*append* ***tn*** **[** **,** ***pn*** **]**

Dyad □*append* writes ***Noun*** as a new component appended at the end of the file whose tie number is ***tn***.

Noun is any noun to be appended as a new component at the end of the file. ***tn*** is the tie number to which the file is tied, followed by a passnumber ***pn*** (if you used a passnumber when you tied the file).

Effect: Provided that the number of bytes the file already occupies does not exceed the file's size limit, a new component is appended. That component contains the noun ⍺ together with its description, which indicates type, rank, and shape. The component also contains control information indicating the number of bytes of storage that the noun requires, the user number of the task that appended the new component, and a timestamp (see the description of □*rdci*).

File permission: 8.

Result: None.

□*appendr*

Append Noun to Tied File with Result

Noun □*appendr* ***tn*** **[** **,** ***pn*** **]**

Dyad □*appendr* writes ***Noun*** as a new component appended at the end of the file whose tie number is ***tn***.

Noun is any noun. ***tn*** is the tie number to which the file is tied, followed by a passnumber ***pn*** (if you used a passnumber when you tied the file).

File permission: 16384.

Result: An *integer item* whose value is the component number of the newly appended component.

⎕arbin

Arbitrary Input with Arbitrary Output

⎕arbout

Arbitrary Output

Monads ⎕arbin and ⎕arbout both cause transmission to the output device of a T-task or the sharer of an S-task. That is all ⎕arbout does; ⎕arbin then awaits one line of input from the device.

Both take arguments in the same form: a numeric list. Both also accept a character argument, and treat it as 0-origin ⎕av⍳⍵.

Both cause transmission to the device from which your APL session is being managed (the screen or printer of a terminal, or the controlling shared name of an S-task, as the case may be). The transmission consists of a sequence of bytes identified by the numbers in the right argument.

When you use ⎕arbout or ⎕arbin in an S-task, the controlling shared name receives a header indicating that what follows is arbitrary output, followed by the characters in ⍵ or, when ⍵ is numeric, ⎕av[256|⍵-⎕io].

When you use ⎕arbout or ⎕arbin in a T-task, the system's session handler transmits a sequence of bytes to the device. The bytes transmitted are 128|⍵, or, when ⍵ is characters, 128|(⎕av⍳⍵)-⎕io.

Values in ⍵ greater than 127 have no effect on what is transmitted through a network such as Reuternet. However, for devices connected directly to a communications controller without intervening network software, the high-order digit of 0 128⊤⍵ controls the high-order bit of the transmission as follows:

0 Even parity.

1 Odd parity.

2 Force the high-order bit to 0 ("space").

3 Force the high-order bit to 1 ("mark").

Even parity is commonly used, and is enforced within Reuternet. The effect of transmitting those characters depends on the physical device to which they are sent.[2]

Arbitrary output is not subject to folding imposed by ⎕pw.[3] The SHARP APL system does not supply a *newline* following arbitrary output. When

[2] Note that since the internal codes used by the APL system do *not* in general correspond to the ASCII or EBCDIC codes used in transmission to devices, in general ⎕arbout 'Hello' does *not* cause Hello to be displayed at the device.

[3] ⎕pw ignores use of ⎕arbin or ⎕arbout, and continues to fold any ⍞-output that may surround the use of ⎕arbin or ⎕arbout as if the arbitrary I/O had not taken place, possibly leading to folds that are misplaced.

the arbitrary output produces a series of characters on the screen or printer, the cursor is left at the position following the last character.

Result: `□arbout`: None.

Result: `□arbin`: A numeric list representing the next transmission received from the terminal or controlling task. In the result, each byte is identified by a number in the range 0 to 255 (for results returned from an S-task or directly from a communications controller) or 0 to 127 (for results transmitted through a network).

The result contains all characters transmitted since the system sent the `□arbin` prompt. Depending on the particular device, that may include characters accumulated in the device's output buffer since the time it was last emptied.[4] The result contains everything up to (and including) the first occurrence of the *carriage return* character, ASCII 13. Unless your device is operating in block mode (so that it resolves cursor movements or overstrikes before transmitting them), the result of `□arbin` contains a record of each keystroke. It is the responsibility of the program that receives this result to decode cursor movements or tabs and to resolve overstrikes.

Whether your terminal displays characters sent during `□arbin` depends on whether the device provides *local echo*. If you use a full-duplex terminal, the system normally echoes transmissions sent to it, but does *not* echo input received by `□arbin`. In such a case your keystrokes produce a visible display only if your APL program echoes the result it receives from `□arbin`.

Watch out: The transmission produced by `□arbout` or by the `□arbin` prompt bypasses the output formatting imposed by SHARP APL. It is quite possible to transmit codes that disrupt the display, or leave the device in a state from which you can no longer control the APL session. Since the prompt may solicit a response directly from the device without interaction from the keyboard, the reply reported in the result of `□arbin` may contain characters the device supplied without your involvement.

[4] Possibly some were accumulated before the prompt was sent.

⎕av

Atomic Vector

The noun ⎕av is a *list* of all possible character values in the SHARP APL system, in numeric order of their internal encoding, from hex 00 to hex FF (decimal 0 to 255).

Their significance as characters in SHARP APL is shown in Figure 15–1, in the previous chapter.

A common use of ⎕av is to provide a reference for translating characters to their numeric equivalents, for example

```
⎕av⍳⍵
```

or to translate from one character encoding to another, by an expression such as

```
NewAlf[⎕av⍳old]
```

To avoid loss of information, *NewAlf* should be a permutation of ⎕av; that is, a list in which each of the 256 possible characters occurs exactly once.

⎕avail

File System Availability

Monad ⎕avail reports whether the APL component file system is available for use. The only acceptable argument is an empty list.

Result: The Boolean item 1 if the file system is available for use. Otherwise, a Boolean item 0.

See Chapter 14, "APL Component Files".

□bounce
Terminate Task

Monad □bounce terminates ("bounces") the tasks identified in ω. ω is a numeric item or list containing the task numbers of tasks running under your user number. The SHARP APL system assigns a distinct task number to each APL task at its start, and displays it as part of the sign-on banner. (□runs returns a table that lists currently running tasks of which you are the owner or initiator.) You may bounce any task running on your user number, and any task which you have started.

Result: A Boolean array having the same shape as ω, with a 1 for each task that was successfully terminated, and 0 elsewhere. The system treats each task as if it had entered the command)continue. That is, it unties any files that the task had opened, and saves a copy of its active workspace. Unless otherwise specified by a previous invocation of □twsid, the workspace of an interactive task (T-task or S-task) is saved under the name continue, and the workspace of a batch task (N-task or B-task) under the name specified when it was started.

Watch out: When you bounce more than one interactive task, each saves a continue workspace. But since these workspaces have the same library number and name (unless overridden by use of □twsid), the last saved overwrites those saved a moment earlier, thereby destroying all but the workspace of the last to be bounced. The order of termination by the system is *not* predictable, due to scheduling decisions made on a moment to moment basis.

Watch out: You may terminate your own task if you include its number in the argument of □bounce.

□cr
Canonical Representation

Monad □cr returns the representation of the most local definition of the verb named in its argument. The result is a character table containing the definition of the verb in *canonical* form; that is, without leading or trailing del symbols (∇) or bracketed line numbers. Each line (including the header) is flush-left. Lines shorter than the longest are right-padded with blanks.

When the argument contains a well-formed name but the visible referent of the name is not an unlocked user-defined verb, the result is an empty table with shape 0 0.

□create
Create a File

fn `□create` ***tn*** `[`***cn*** `[,`***lpn***`]]`

Dyad `□create` creates a new file with name ***fn***, and share-ties it with ***tn***. ***tn*** must be a positive integer less than 2^{31} that is *not* now in use as a tie number. ***cn*** is an optional positive integer less than 2^{31} to be used as the starting component number for the file being created.

lpn is an optional library passnumber. If the library passnumber is specified, the starting component number must be supplied as well.

For example:

```
      '12345 michigan' □create tfn, 69
      □size tfn
69 69 0 102620
```

fn is a character list containing a file name that is both valid (that is, well formed) and available (that is, not already in use in that library). Optionally, the file name is preceded by a library number and followed by a file size limit; for example:

```
'1431860 DataFile 125000' □create tn
```

A valid name is formed in the same way as the name of an APL noun or verb, but may not exceed 11 characters nor contain ⍺ or ⍵.[5] A name may include letters from either alphabet or the characters Δ or ∆, and numerals `0` to `9` except as the first character.

If you omit the library number, the file is presumed to belong to your own library. You may explicitly include the library number as characters preceding the name and separated from it by a blank. The library number must be your own or the number of an already-existing public library (one having a library number less than 1000), or you must have *create* library permission for the library in question.

If you omit the file size limit, a site-defined limit is assumed. If you include a limit, it must be written as characters in integer format without commas (for example, `2000000` rather than `2,000,000` or `2e6`).

Effect: Provided the arguments are valid, a new file is created, share-tied to the tie number indicated in the right argument. The new file has no components. It has a 0-by-3 access table, giving the file's owner *unlimited access* to it, and no access to anyone else. See the sections headed "Library

[5] The rules for file name formation depend on the host system. In SHARP APL/PC, for example, file names must be 8 characters or less in length, and may not contain non-ASCII characters.

Access Controls" and "File Access Controls" and the ensuing discussion of the access table in Chapter 14, "APL Component Files".

File permission: None.

Library Permission: `1`.

Result: None.

□ct
Comparison Tolerance

The system noun □ct establishes the maximum relative difference allowed between two numbers if they are to be considered equal. Comparison tolerance permits the interpreter to ignore small differences between numbers likely to arise from inexact internal representations that are inherent in computer arithmetic on numbers that are not integers.

Some numbers are inherently incapable of exact representation (for example, π, e, $\sqrt{2}$). Other inexact representations arise because input and display are normally in base 10, but internal representations are in base 16. For example, the representation of 0.1 in binary is an endless sequence of binary digits, even though it is finite (and short) in base 10. Yet others arise from rounding errors, when numbers that do in principle have finite representations must be truncated to fit within the system's word size. Tolerant comparison permits reasonable identities to work even when the system is incapable of representing them properly. Thus, the expression

```
1 = 3×÷3
```

is reported as true, despite the fact that 3 times the reciprocal of 3 does not really produce an exact 1.

In a clear workspace, the default value for □ct is `1.41969769e¯14` (computed as `1023×16*¯14`). You may assign the visible □ct to be any numeric item not less than 0, and less than 16^{-8}. Before you set □ct to zero, it may be rewarding to consider the effects of intolerant comparison in the example shown in Figure 16–2.

A way to get exact comparison without setting □ct to zero is by comparison with zero. Instead of `x=y` with `□ct←0`, use `0=x-y`.

Verbs Affected by □ct

The visible value of □ct is part of the definition of the following verbs when an argument is represented internally in floating point or complex form.[6]

[6] The maximum permitted value of □ct is chosen so that □ct could never affect the outcome of any of these verbs when applied to the arguments that could be internally represented as integers.

Dyads: | ∧ ∨ ∈ ⍳ < ≤ = ≡ ≠ ≥ > ∈

Monads: ⌊ ⌈ ≠

⎕*ct* is a relative comparison tolerance. When two numbers are compared and at least one of them is a floating-point number, their relative difference (which depends on their magnitudes) is compared against ⎕*ct*. Two numbers are considered equal if their relative difference is less than or equal to ⎕*ct*. That is, ⍺ and ⍵ are judged equal when:

```
(|⍺-⍵) ≤ ⎕ct×(|⍺)⌈(|⍵)
```

When one of the arguments (say, ⍵) is 0 but the other is not, that inequality reduces to:

```
(|⍺) ≤ ⎕ct × |⍺
```

Since ⎕*ct* must be (much) less than 1, the inequality can never be satisfied. Thus the value of ⎕*ct* can never affect a comparison with 0.

When ⎕*ct* is 0, all comparisons are exact. The judgment "not equal" may be produced by bits that would otherwise be considered insignificant.

Figure 16–2 shows how the results of some simple expressions depend upon the value of ⎕*ct*.

eps←1e¯14	*a* *b*	0 0+*eps*	0 0−*eps*	1 1+*eps*	1 1−*eps*
⎕*ct*←0	⌊*b*	0	¯1	1	0
⎕*ct*←10×*eps*	⌊*b*	0	0	1	1
⎕*ct*←0	⌈*b*	1	0	2	1
⎕*ct*←10×*eps*	⌈*b*	0	0	1	1
⎕*ct*←0	*a*=*b*	0	0	0	0
⎕*ct*←10×*eps*	*a*=*b*	0	0	1	1
⎕*ct*←0	*a*<*b*	1	0	1	0
⎕*ct*←10×*eps*	*a*<*b*	1	0	0	0
⎕*ct*←0	*a*⍳*b*	5	5	5	5
⎕*ct*←10×*eps*	*a*⍳*b*	5	5	3	3
⎕*ct*←0	*a*∈*b*	0	0	0	0
⎕*ct*←10×*eps*	*a*∈*b*	0	0	1	1

Figure 16–2 : Tolerant and intolerant comparisons.

The following illustrate the effects of □*ct* in a few simple examples:

```
      □ct
1.419697693e¯14

      ⊢a←*⊗6
6

      a=6
1

      ⌈a
6

      □ct←0
      a=6
0

      ⌈a
7

      a-6
¯2.220446049e¯16
```

□*dl*

Delay Execution

Monad □*dl* delays execution of the statement in which it appears. ω is a numeric item, specifying the duration of the desired pause in seconds. The result is a numeric item, the actual delay in seconds.

When the delay completes normally, the result is not less than ω. The actual delay (reported as the result) may be slightly more than requested. For example:

```
      □dl 10
10.033
```

(Appears about 10 seconds after pressing *Enter* on the line above.)

Behavior of Delay When Interrupted

The delay produced by □*dl* can be cut short by a weak interrupt. As usual, a weak interrupt halts execution at completion of the end of the line being executed. The weak interrupt causes □*dl* to return its result without further waiting. When the system has completed execution of the entire

line, execution halts before starting a new line. (A weak interrupt on a line containing multiple instances of `⎕dl` cuts short all those that have not yet delayed.)

Since a weak interrupt causes an immediate return from `⎕dl`, a definition for strong interrupt does not arise.

⎕drop

Drop File Components

Monad `⎕drop` ω drops components from the beginning or end of a tied file. The argument is:

`⎕drop` ***tn*,*n*** **[*,pn*]**

tn Tie number of a tied file.

n Number of consecutive components to drop. A positive number counts from the beginning, a negative number from the end of the file.

pn The passnumber, if the file was tied with a passnumber.

Effect: The indicated number of components is dropped from the file. You can drop a block of consecutive components either from the beginning or from the end of a file. That reduces the range of valid component numbers. Suppose a file has 100 components, numbered from 1 to 100. If you drop the last 10, it will then have 90 components, with numbers running from 1 to 90. But if you drop the first 10, it will have 90 components numbered from 11 to 100. Dropping components from a file does *not* alter the numbers by which you refer to the components that remain.

If you attempt to drop more components than exist, the system rejects the expression with the message `file index error`.

File permission: `32`.

Result: None.

See Chapter 14, "APL Component Files".

□ec
Environment Condition

The system noun □ec is part of the event trapping apparatus. Its value is restricted to a Boolean item. When its value is 1 *at any level of the state indicator*, an *untrapped* error in a defined verb causes an exit from the verb (whereas otherwise execution would halt). However, a stop control[7] still halts execution in the usual way.

As illustrated in the right side of Figure 16–3, when □ec is set to 1, upon encountering an error, SHARP APL exits from *foo* and attributes the error to the verb *foo* as a whole, rather than to a specific statement within it.

Example with □ec set to 0:

```
   ∇ foo; □ec
[1]  □ec←0
[2]  2+ ⍴     Error
   ∇

      foo
syntax error
foo[2] 2+ ⍴    Error
        ^
```

Example with □ec set to 1, or localized but not set:

```
      ∇ foo; □ec
[1]    □ec←1 ⍝Can omit this
[2]    2+ ⍴    Error
    ∇

       foo
syntax error
       foo
       ^
```

Figure 16–3 : Behavior of errors depending on □ec setting.

□ec is a *system shared noun*. That is, it is shared with the SHARP APL system, and the system may set its value. In particular, whenever SHARP APL starts execution of a verb whose definition localizes the name □ec, SHARP APL immediately sets □ec to 1. As a consequence, when □ec appears in a verb's header, there is no way to interrupt it in a way that leaves it suspended until the verb itself has set □ec to 0.

When □ec has the value 1 (either because it has been explicitly set, or because SHARP APL has set it because it is localized), any verb invoked within the verb to which □ec is local is also affected. Moreover, this extends to *all* verbs that may be invoked, regardless of level, and regardless of whether they localize □ec. Once an outer verb (that is, one that appears lower in the state indicator) has localized □ec, a verb can halt only when:

- The visible value of □ec *and also* the value of □ec at *all* levels of the state indicator are 0

[7] For a verb *v*, the expression *s*Δ*v*←*n* sets stops on the lines of *v* whose line numbers are members of *n*.

or

- A stop control is set for one of the verb's lines and event 1001 (*stop*) is not trapped.

See also Chapter 10, "Event Handling".

□er
Event Report

This noun is set by the interpreter whenever an error, interrupt, or other such event occurs during the execution of a program. You (or the recovery expression contained in □*trap*) may then examine □*er*, perhaps just to display it, or to decide on remedial action.

When □*er* is localized, the □*er* that is set is the one visible at the level at which the event is handled. When the event is trapped and the recovery expression uses action *c* to cut back the stack, or when □*ec* is 1 so that a halt is not permitted, the error report appears not in □*er* of the level at which the event originally occurred (which may no longer exist), but at the level of the effective trap (when the event is trapped), or the level at which execution halts (when the event is not trapped).

In general, □*er* is a table with three rows. It has a fourth row when the event is 6, *value error*, *and* the event has been trapped with action *i* (*immediate*). The rows contain:

□er[1;⍳4] The event number, in four characters, right justified. Figure 16–4 contains a list of the event numbers that are currently generated by SHARP APL.

5↓□er[1;] The event title. For events signaled by SHARP APL, this is the standard name associated with the event (see Chapter 10, "Event Handling"), as modified by the current setting of)*nlt*. An event signaled by □*signal* may have its own event title, specified by the left argument of □*signal*.

□er[2;] The sentence being executed when the event occurred.

□er[3;] Caret pointing to the part of the line being executed when the event occurred.

□er[4;] The name that lacks a value (when event 6 has been trapped with action *i*).

When the workspace is so full that there is insufficient room for an appropriate □*er*, SHARP APL makes □*er* a four-element list containing only the character representation of the event number.

Errors
0 Any error*
1 Workspace full 2 Syntax error 3 Index error
4 Rank error 5 Length error 6 Value error
7 Format error 8 Result error 9 State indicator error
11 Domain error 15 Symbol table full 16 Nonce error
18 File tie error 19 File access error 20 File index error
21 File full 22 File name error 23 File damaged
24 File tied 25 File system hardware error 26 File system error
28 File system not available 29 File library not available 30 File system ties used up
31 File tie quota used up 32 File quota used up 33 File reservation error
34 File system no space 38 File component damaged 45 Workspace locked
46 Workspace not found 47 Workspace not readable
67 Swap error, clear workspace** 68 System error, clear workspace**
72 Interface quota exhausted 73 No shares 74 Interface capacity exceeded
75 Share table full 76 Processor table full 77 Identification in use
78 □*pp* error 79 □*pw* error 80 □*io* error
81 □*rl* error 82 □*ct* error 83 □*ht* error
84 □*ps* error 85 □*fc* error 95 Workspace not writable
Interrupts
1000 Any interrupt***
1001 Stop 1002 Attention 1003 Interrupt
1004 Input interrupt 1005 Shared name interrupt 1006 File interrupt 1007 File backup interrupt
Workspace State
2001 Return to immediate execution****

* 1–999; 0 never occurs in □*er*.

** No way to respond, but – briefly – reported in □*er*.

*** 1001–1999; 1000 never occurs in □*er*.

**** Not reported in □*er*.

Figure 16–4 : Event numbers.

⎕erase

Erase a Tied File

fn `⎕erase` ***tn*****`[,`*****`pn`*****`]`**

Dyad `⎕erase` erases the file tied to ***tn*** whose name is ***fn***, with optional passnumber ***pn***.

For example, if file `1431860 DataFile` was tied to `dfn`, it could be erased by the expression:

```
'1431860 DataFile' ⎕erase dfn
```

The right argument contains the tie number of a *tied file*. If you specified a non-zero passnumber when you tied the file, it must appear as the second element of the right argument.

The left argument is a character item or list containing the name of the file to be erased. The left argument is redundant: all this information is available from the tie number. However, The system requires you to provide both forms of identification to make it less likely that you will inadvertently erase the wrong file.

Effect: The file ceases to exist. The number formerly tying it is again free for use. The file's name disappears from the result of `⎕nums`, of `⎕names` and of `⎕lib`. The SHARP APL system marks the space the file occupied as available for other use.

File Permission: `4`.

Result: None.

⎕ex

Expunge Objects

Monad `⎕ex` expunges (that is, erases) from the active workspace the most local referent of each of the names in its argument. ω may contain zero or more names, arranged as a table with one name per row, or as a character list or item when there is only one name.

The result is a Boolean list, with one element for each name in the argument. Each element of the result contains:

- `1` When the corresponding name is now free (either because it had no referent or because its former referent has been erased).
- `0` When the corresponding name is not free (either because it has a referent that cannot be erased, or because it is not a well-formed name).

`□ex` does not erase a name whose referent is a label or a system noun or verb. A verb that is executing, pendent, suspended, or waiting cannot be erased.

If `□ex` produces a *ws full* or *domain error*, nothing has been erased.

□fc

Format Control

The system noun `□fc` acts as a global argument to dyad ⍕ with a character left argument (*format by example*). It specifies how ⍕ deals with certain cases. `□fc` is a six-item character list. The following list shows the value assigned in a clear workspace, and the significance each item has in formatting:

`□fc[1]` . Decimal point. Character to separate the integer part from the fractional part of a fraction. This character replaces . where it appears in ⍺ as a decimal point.

`□fc[2]` , Grouping separator. This character replaces , where it appears in ⍺ with digits on both sides.

`□fc[3]` * Fill to represent insignificant digits at positions marked in ⍺ by an `8`.

`□fc[4]` * Character to fill an overflow field (one too small to contain the representation of a number in ⍵). When `□fc[4]` is the character `0`, an overflow in any field causes SHARP APL to reject the entire expression ⍺⍕⍵ as a *domain error*.

`□fc[5]` _ Marker for blank. Where this character appears in ⍺ as a controlled character (that is, embedded within the positions devoted to the number representation), the result contains a blank.

`□fc[6]` ¯ Proposed as character in the result to indicate a negative value. (Not implemented.)

For more information on ⍕, see Chapter 6, "Verbs".

⎕fd

Verb Definition

Dyad ⎕fd provides services for conversion between a verb and its character representation. ⍺ is a numeric item and selects one of the services described below. Its value must be a member of the set `1 2 3 6 7`.

`1 ⎕fd ⍵` ***List representation.*** ⍵ is the name of a user-defined verb, as a character list. The result is a character list (including *newline* characters at the line boundaries) showing the definition the way it looks when displayed by the ∇-editor: with line numbers enclosed in brackets, and a leading and closing ∇. When ⍺ is not the name of a verb, or has a locked definition, the result is an empty list.

`2 ⎕fd ⍵` ***Table representation.*** ⍵ is the name of a user-defined verb as a character list. The result is a character table containing the *canonical representation* of the verb's definition, with one row for each line of the definition. The lines are shown the way they look when displayed by ⎕cr: flush-left, without line numbers, and without the opening and closing ∇. When ⍺ is not the name of a verb, or has a locked definition, the result is an empty table.

`3 ⎕fd ⍵` ***Fix a verb from its representation.*** ⍵ is the character representation of a user-defined verb. It may be in either the table form returned by ⎕cr or `2 ⎕fd`, or the list form returned by `1 ⎕fd`, with successive lines separated by a *newline* character, with or without line numbers, with or without opening and closing ∇. Blank lines are permitted within the representation.

Side effect: The verb defined by ⍵ is *fixed* (that is, materialized in the workspace) in the same manner as ⎕fx. A definition can be fixed provided the name shown in ⍵ is available: that is, has no visible use, or its visible referent is to a verb (in which case the new definition replaces the old).

When the name of the verb to be fixed has been localized in some verb that is currently on the state indicator stack, the newly defined verb is local to that verb. It ceases to exist when the verb to which its name is local completes execution.

Result: The name of the verb thus fixed. When for any reason the definition cannot be carried out, the result is a two-element numeric list whose first element is a diagnostic code and second the line number of the line in error. The diagnostic codes are:

1 Workspace full
2 Definition error
3 Character error
4 Symbol table full

`6 □fd ω` ***Expunge names.*** ω is a *namelist*, that is, either a list with successive names separated by blanks, or a table with one name per row. Each of the names in ω is freed: that is, its visible referent is expunged so that the name is free for other uses. The result is a table containing those names that were *not* freed (because their referents cannot be expunged or because they were not well formed). Thus when `6 □fd` is successful, the result is an empty table.

`7 □fd ω` ***Lock verbs.*** ω is a *namelist* (as defined above). Each of the verbs named in ω is locked. The result is a table containing those names that were *not* locked (because their visible referents were not verbs). Thus when `7 □fd` is successful, the result is an empty table.

□fhold

Hold Tied Files

□hold

Hold Tied Files

These verbs coordinate concurrent access to shared files. They permit one APL task to request temporary exclusive use of all or part of a file that is at the same time share-tied by other tasks.

`□fhold` is the newer and more general form. `□hold` is unable to accept a left argument, and is unable to include passnumbers or component ranges in its right argument. However, for uses that do not require these, `□fhold` and `□hold` have the same effect. In particular, they operate on a common queue of held files, and in that respect are interchangeable.

ω is an array of tie numbers.

For `□fhold`, ω may also be a table of one to four rows. When α is a four row table, the rows are used as follows:

`ω[1;]` Tie numbers. A one-row table has the same effect as a list.

`ω[2;]` Passnumbers.[8] Passnumber `0` is equivalent to "no passnumber."

`ω[3;]` Lower bound of a hold range.

`ω[4;]` Upper bound of a hold range. The effective hold range includes the lower and upper bounds. Thus, if the lower and upper bounds are the same, the range is one component. If the lower bound is greater than the upper bound, the range is empty.

When ω has fewer than four rows, the following defaults apply:

3 rows: The hold range is one number, identified in row 3.

2 rows: The hold range is not specified, and applies to all components.

1 row: No passnumbers (equivalent to passnumber `0`). Hold applies to all components.

The *hold* mechanism permits cooperating applications to pass information to each other by way of a file that both have tied. Where they wish to hold specific ranges within a file, the application seeking to hold a range executes `⎕fhold` with a three-row or four-row argument. The third and fourth rows contain the first and the last of a range of consecutive numbers. When no other task has executed `⎕fhold` with respect to the same file *and* a range that overlaps the range stated in ω, `⎕fhold` completes execution. (It does not return a result.)

But when another task has executed `⎕fhold` for the same file and a range that overlaps the range stated in ω, SHARP APL delays until such time as the other tasks release all holds that overlap the range stated in ω. (See below, under "Monads `⎕fhold` and `⎕hold`" or "Dyad `⎕fhold`", as appropriate.)

When no component range is stated, the hold applies to the entire range of integers -2^{31} through $2^{31}-1$ (and thus to all possible components when the range is interpreted as a block of consecutive components). The components described by the range need not exist. This allows `⎕fhold` to be used as a general enqueue mechanism.

File Permission: `2048` for `⎕fhold`; `64` for `⎕hold`.

Result: None.

8 Because `⎕hold` is unaffected by passnumbers, a file used for an application requiring passnumbers is usually given an access table prohibiting use of `⎕hold`.

Monads □*fhold* and □*hold*

1. All preceding holds (if any) are released. It does not matter whether the new request to hold applies to the same files or to different ones. Each execution of either of the hold verbs cancels any preceding use. (For more controlled release, see dyad □*fhold*, below.)
2. Execution of your task goes into an indefinite delay until all the files you have asked to hold are free of holds set by other tasks.

   From the keyboard, you can signal an interrupt. Interrupt cancels a pending hold even when the interrupt is trapped. If the interrupt is not trapped, control returns control to the keyboard.
3. Once all the files you have requested are free, your hold takes effect. Execution of your task proceeds.[9]

When the interpreter receives from another task a request to hold any of the files you are holding, or for a range that overlaps a range you are holding, that task goes into an indefinite delay until either you release your hold on the files the other task has asked for, or you untie them. SHARP APL does not inform you of a pending request by another task.

File permission: `2048` for □*fhold*; `64` for □*hold*.

Result: None.

Dyad □*fhold*

⍺ is an item, either `1` or `¯1`, with effects as follows:

`1` ***Selective hold*** The files identified in ⍵ are added to the hold queue without releasing any files previously held. If a file is currently held, and it is also specified as one of the argument files in `1` □*fhold*, the ranges specified must match those already being held, or a *domain error* will occur.

`¯1` ***Selective release*** The files identified in ⍵ are released without releasing any other files.

The result of dyad □*fhold* is either a *list* of the tie numbers of the files held following its execution, or a three-row table showing the tie numbers of the held files together with the first and last members of the held range of each file.

When ⍵ has two rows or fewer (that is, refers to files but not to held ranges), the result is a list, with one item for each held file, containing their tie numbers.

[9] See "Duration of a File Hold", below.

When ⍵ has three or four rows, the result is a three-row table, in which the first row contains the tie numbers of held files, and rows 2 and 3 contain the first and last members of their held ranges.

When ⍵ is empty, there is no change to way files are held, so any of

`1 ⎕fhold ''` or `1 ⎕fhold 1 0⍴''` or `1 ⎕fhold 2 0⍴''`

returns a list of the tie numbers of held files, and either of

`1 ⎕fhold 4 0⍴''` or `1 ⎕fhold 3 0⍴''`

returns a table showing the tie numbers and ranges of all holds.

File Permission: `2048`.

Result: A list of tie numbers for a right argument of two or fewer rows, or a table of tie numbers and ranges if the right argument has more than two rows.

Application to a Range of Numbers

When ⍵ has three or four rows, the hold applies only to a specific number (three rows) or to a specific range of numbers (four rows). When two tasks make requests to hold the same file, the requests interlock when there is any overlap of the ranges requested, but not otherwise.

The tasks may interpret the two numbers as a block of consecutive component numbers. However, there is no requirement to interpret them that way, or that components with those numbers actually exist. Indeed, *any* numbers internally representable as integers (that is, from -2^{31} through $2^{31}-1$) may be held. Thus the `⎕fhold` mechanism may also be used as a signal between tasks. It is not essential that they make any use of the file, or that the file have any components – but they *do* have to tie it.

Duration of a File Hold

A file hold, once in effect, is ended by any of the following:

- `¯1 ⎕fhold` for that file.
- Any use of monad `⎕fhold` or `⎕hold`.
- The end of your task (sign-off, bounce, crash, etc.).
- Untying the affected file or files.
- Any untrapped error or interrupt.
- Any return to immediate execution (including one trapped as event 2001, since the only actions possible with event 2001 lead ultimately to immediate execution). However, returning to the keyboard for ⎕- or ⍞-input does *not* terminate a hold.

Watch out: You cannot experiment with file holds by entering statements one after another in immediate-execution mode. Any hold you enter that way terminates immediately when the system returns for the next entry from the keyboard. In that situation, the most you could do would be to put into a single line of entry (separated by diamonds) both the file hold and the statement you want executed during the hold.

Watch out: Inside the block spanned by a file hold and its subsequent release, make sure than none of the verbs invoked there issues a monadic hold, since it would thereby undo your earlier file hold when it issued its own.

See also the discussion in Chapter 14, "APL Component Files".

□*fi*

Input Format Conversion

Monad □*fi* returns the numeric values of the fields in a character list. It assumes that within ⍵ fields are delimited by blanks.

The result is a numeric list containing the value of each well-formed numeric field, and 0 for each other field.

```
      □fi '666 ¯1.20  -5.5    .1  314159e¯5   five'
666 ¯1.2 0 0.1 3.14159 0
```

The value -5.5 is invalid because it contains a minus sign - instead of a negative sign ¯.

□*fi* accepts any of the forms that may appear in default system output (or in the result of monad ⍕). That includes integers, decimal fractions, *e*-formats, complex numbers, and numbers preceded by ¯ to indicate negative values. Those are the only forms □*fi* recognizes; in particular, it does not consider well-formed a field that contains a currency symbol or embedded commas, or that is enclosed in parentheses, and returns 0 for each such field.

```
      □fi 'Take 3 spoons of 150 proof rum'
0 3 0 0 150 0 0

      □fi 'Sent $3.00 for 6 kilos on 87/4/1'
0 0 0 6 0 0 0
```

To permit □*fi* to convert some common but otherwise unacceptable formats, techniques such as the following may be useful:

```
      ext←' 0123456789e.,-¯'
      int←' 0123456789e. ¯¯*'
      ⎕fi int[ext⍳' .25 -6.25 8,9,10']
0.25 ¯6.25 8 9 10
```

Use of ⎕*fi* with ⎕*vi*

Since ⎕*vi* returns a Boolean result indicating which fields were well formed, the expression

```
(⎕vi ⍵)/⎕fi ⍵
```

returns only the values of the well-formed numbers.

⎕*fm*

Function Monitor

⎕*fm* collects and reports measures of frequency of use, CPU time and elapsed time during the execution of user-defined verbs.

Dyad ⎕*fm*

Dyad ⍺ ⎕*fm* ⍵ sets or reports monitoring. ⍵ is a character array of names, each name in a 1-cell of ⍵ (that is, arranged along the last axis). Commonly, ⍵ is a table with one name per row.

⍺ is a numeric item indicating the data to be collected or reported. It applies to all the verbs named in ⍵. The sign of ⍺ indicates whether you are establishing monitoring or reporting data already collected:

⍺ ***non-negative*** Establishes which data should be collected during future executions of the affected verbs.

⍺ ***negative*** Establishes which topics should be reported from the data already collected.

The topic (collected or reported) is represented by the sum of the codes for the various possible topics. Since each code is a power of 2, each topic can be enabled or reported independently of the others.

⎕*fm* acts on the visible (most local) definition of each verb. In case of an error during the execution of ⎕*fm*, the state of the workspace remains unchanged.

Turning on any form of monitoring for a verb turns off any previous monitor setting for that verb. Turning on any monitoring other than *summary* automatically generates summary monitoring.

The numeric codes for the various monitoring reports are as follows:

`0`		***No monitoring.*** When ⍺ is zero, this (a) discontinues data collection, and (b) discards data from previous monitoring and frees the space used for its collection.
`1`	`2*0`	***Summary monitoring.*** Records the monitoring information for the topics associated with codes 2 through 128 for the verb as a whole, without line-by-line breakdown.
`2`	`2*1`	***Execution count.*** Records the number of times each line was executed.
`4`	`2*2`	***CPU/own.*** Records total CPU time on each line, but excluding user-defined verbs that the line invokes.
`8`	`2*3`	***Elapsed/own.*** Records total elapsed time on each line, but excluding user-defined verbs that the line invokes.
`16`	`2*4`	***CPU/total.*** Records total CPU time on each line, including user-defined verbs that the line invokes.
`32`	`2*5`	***Elapsed/total.*** Records total elapsed time on each line, including user-defined verbs that the line invokes.
`64`	`2*6`	***CPU/non-monitored.*** Records CPU time for user-defined verbs that the line invokes and are not themselves monitored.
`128`	`2*7`	***Elapsed/non-monitored.*** Records elapsed time in those user-defined verbs that the line invokes but are not themselves monitored.

The result of dyad ⎕*fm* is an array with a cell for each name in ⍵ (that is, its frame-shape is `¯1↓⍴⍵`). The cells are different, depending whether ⍺ is 0, positive, or negative, as follows:

⍺ ***is zero***	A result cell is an item, reporting the former level of monitoring for each name in ⍵.
⍺ ***is positive***	A result cell is an item, reporting the new monitoring status for each name. When the request to set monitoring is successful, each item in the result has the same value as ⍺. When the system is unable to establish the monitoring requested (for example, because a name in ⍵ does not refer to a visible user-defined verb), the corresponding element of the result is `0`.

α *is* ¯1 A result cell is a seven-item list. The items within that list are the summary values for each of the reports corresponding to codes `2*1 2 3 4 5 6 7`.

α *is even and negative* Each result cell is a table, containing a line-by-line report, with a row for each line of the verb. (For this reason, ω may ask for a report on more than one verb if and only if the verbs have the same number of lines. A request for reports on verbs with different numbers of lines is rejected as a *length error*.)

Each result cell has a column for each of the topics requested in α. A request for a report on a topic for which monitoring has not been enabled is rejected as a *domain error*.

The values reported both for CPU time and elapsed time are in seconds. Their precision varies somewhat with the CPU on which they were recorded, but they are generally precise to within a microsecond. The values for CPU are not "billing units" – that is, they are *not* adjusted by the CPU multiplier used in calculating billing.

Monad □*fm*

ω is a table of names in the same manner as for α □*fm* ω.

The result frame has shape ¯1↓ρω. Each cell is a numeric item indicating the monitoring status of a name, encoded in the same way as the left argument of dyad □*fm*.

Storage Allocated to Monitoring Data

Turning on the monitor for a verb (by α □*fm* ω with a positive left argument) causes the interpreter to allocate additional space in the verb definition, in order to accumulate the requested data. The additional space is included in the result of `4 □ws`.

When you save a workspace, a verb's monitoring data is saved with it, and is loaded when the workspace is subsequently loaded, so that monitoring may span *)save* and *)load*.

However, copying, editing, fixing, or defining a definition[10] turns off monitoring and discards the space formerly allocated for data collection. When you pack the definition of a verb, monitoring is similarly discarded

10 That is,)*copy*, ∇-editor, □*fx* or `3 □`*fd*, □*pdef* or □*ppdef*.

from the definition in the package. (The original verb is not changed; only the copy created inside the package.)

You may set monitoring on a verb that is pendent or suspended. Monitoring takes effect at once, but (naturally) lacks data on the portion already executed.

When an event is trapped, the cost of executing the recovery expression is charged to the line that was active when the event was trapped.

You request monitoring of a verb whose definition is locked, but only with a left argument of `¯1` (summary). If you lock the definition of a verb for which monitoring was already in effect, monitoring is thereby turned off (but you may then request summary monitoring).

Monitoring a Recursive Definition

You can monitor a recursive or re-invoked definition. The total CPU and elapsed time spent on lines including invoked verbs is equal to the time spent on the root verb. Suppose you have a verb `foo` that calls `foo` that in turn calls `foo`. You can think of that as though the invoked versions are different verbs: as though `foo` calls `foo2` that calls `foo3`, and you are only recording data for `foo`.

When there is *no* recursion, the column for *Total CPU*, including invoked verbs, is equal to the sum of `¯16 ⎕fm ⍵`. When there *is* recursion, the sum of `¯16 ⎕fm ⍵` is greater than the actual CPU time consumed by the verb.

Similarly, when there is *no* recursion, the column indicating total elapsed seconds, including invoked verbs, is equal to the sum of `¯32 ⎕fm ⍵`. When there *is* recursion, the sum of `¯32 ⎕fm ⍵` is greater than the actual elapsed time consumed by the verb.

□fmt

Format Output

Dyad □fmt formats character and numeric data into a character table. The conventions for the left argument of □fmt resemble certain of those used in Fortran or Cobol.

Format Phrases

α is a character list divided into *format phrases* separated by commas. See Figure 16–5 for a summary of format phrases. For example, the following contains three phrases:

```
f←'i6,f8.2,g< 999.99>'
```

Each format phrase governs the appearance of a column of ω. The number of phrases need not match the number of columns in ω; the phrases are repeated cyclically as needed.

Each phrase has a *type*, indicated by a letter,[11] such as the letters i, f, or g in the preceding example.

A *constant* field is reproduced in the output but does not correspond to any column of the data. A constant phrase is enclosed by the characters <> (or also by certain other characters), thus:

```
ff←'<Part: >,i6,< Cost: >,f8.2'
```

A *tab* phrase does not itself control a field of data, but controls the position in the result at which the next field starts, specified either as a *relative* distance forward or back from the end of the previous field, or as an *absolute* location in the result.

11 The control letters are all from the first alphabet.

n		`a`	***w***	***Character data.*** Print each character in a field ***w*** positions wide. (Note that when ***w***>1, this inserts blanks between adjacent letters.)
n	***q***	`e`	***w.d***	***Exponential fraction.*** Print in a field ***w*** positions wide, with ***d*** significant digits.
n	***q***	`f`	***w.d***	***Fixed-point fraction.*** Print in a field ***w*** positions wide with ***d*** decimal digits. Trailing zeros always print.
n	***q***	`i`	***w***	***Integer.*** Leading zeros are represented by blanks unless the `z` qualifier is used.
n		`x`	***p***	***Relative reposition ("tab").*** Start the next field at the position displaced ***p*** positions from the end of the preceding field. The displacement may be negative, in which case the next field may overwrite earlier ones.
n		`t`	***p***	***Absolute reposition ("tab").*** Start the next field at position ***p*** (possibly over-writing preceding fields).
			<text>	***Constant.*** Insert ***text*** in every row of the result.
n	***q***	`g`	***<text>***	***Picture.*** Insert the digits that represent the value (rounded to the nearest integer) into the "picture" text. Build the result as follows: Copy ***text*** to the field. Then overlay the successive digits of the number's representation. Place the digits at the positions marked in the picture by `9` or `z`, starting with the low-order digit in the right-most `9` or `z`. In positions marked `z`, represent leading or trailing zeros by blanks. Where other text characters are embedded between digits used in the result, keep them. Overlay right decorators to the right of the right-most digit used in the result, and left decorators to the left of the left-most digit used in the result.
				Key:
n				Number of repetitions (optional).
	q			Qualifier (see Figure 16–6).
			w	Field width.
			d	Number of decimal digits (`f` format); number of significant digits (`e` format).
			p	Number of positions to displace next field.

Figure 16–5 : Summary of `⎕fmt` *format phrases.*

Repetition Factors

A phrase or group of phrases may be preceded by a repetition factor. For example,

```
g←'2i6,3f8.2,g< 999.99>'
```

indicates that the first phrase (`i6`) is to be used for two columns, the next (`f8.2`) for three columns, and the last (`g< 999.99>`) for one.

The repetition factor can apply to a set of consecutive phrases enclosed in parentheses. For example,

```
g←'2(i6,3f8.2),g< 999.99>'
```

indicates that the sequence of one column formatted by `i6` and three columns formatted by `f8.2` is to be repeated twice.

Type and Shape of the Right Argument of ⎕*fmt*

ω is a numeric array or character array of rank no greater than 2, or list of one or more boxes containing such arrays. When ω is a list or item, it is treated as if it were the one-column table $⍪\omega$.

When ω is formed as a succession of boxes, each is disclosed, treated as a table, and its successive columns formatted in order from left to right. Columns need not have the same height; short columns in the result are padded with blank rows at the bottom to match the length of the highest.

Shape of the Result of ⎕*fmt*

The result returned by ⎕*fmt* is always a character table. The number of rows is sufficient to accommodate the tallest column in ω.

In the result, successive bands of adjacent columns form *fields* corresponding to the phrases in the left argument α. Each field has the width specified by the corresponding phrase in α. There are no additional columns between fields, so the visual separation of fields in the displayed result is produced either by constant phrases in α, or by specifying formats that are wider than needed to represent the data.

Qualifiers and Decorators

Within each format phrase, to the left of the letter that identifies its type, you may insert various qualifiers or decorators. A *qualifier* (See Figure 16–6) provides additional rules, for example "insert commas between triplets of digits" or "leave this field entirely blank when the value being represented is zero."

A *decorator* (See Figure 16–7) is a piece of text attached to the representation of a number, usually to indicate its sign. Decorators precede the format phrase with which they are associated. When used, decoration text appears to the right of the right-most digit used in the representation, or to the left of the left-most. For example, to enclose negative numbers in parentheses, you need to include in the format phrase the decorators *m* (the left text of a negative number) and *n* (the right text of a negative number, thus:

```
      'm<(>n<)>f8.2' ⎕fmt ¯123.45
(123.45)
```

To preserve right-alignment, when you include decorators that appear to the right, you need to specify equal-length decorators for negative and non-negative numbers, thus:

```
      'm<(>n<)>p< >q< >f8.2 ⎕fmt 123.45 ¯123.45
 123.45
(123.45)
```

Alternatively, you may exploit decorators of different size to give different alignment to negative and non-negative values:

```
      'm<->q<        >f16.2' ⎕fmt 1 ¯1 1 ¯1 × 123.45
123.45
         -123.45
123.45
         -123.45
```

The background decorator *r* first fills the field by repeating as necessary the character(s) specified, and then overwrites the field with the digits of the representation and the sign decorators. The effect is to leave the background text visible only in the positions unused by the representation. This is commonly used for "check protection," as follows:

```
      'r<$**********>f12.2' ⎕fmt 12345.67 5.67
$**12345.67
$******5.67
```

`b`	***Blank.*** Make the entire field blank when the value is zero.
`c`	***Commas.*** Insert commas between successive triplets of digits. (May not be used with `g` format.)
`ke`	***Scale.*** Scale the result by displaying a value 10^{e} times the value in ω.
`l`	***Left justify.*** Left justify the representation within its field. (May not be used with `g` format.)
`z`	***Zeros.*** Print leading zeros or trailing zeros. (NB: In `g` format, `z` within the picture makes leading or trailing zeros blank.)

Figure 16–6: Summary of `⎕fmt` *qualifiers.*

`m<`***text***`>`	***Negative/Left.*** Insert ***text*** to the left of the representation of a negative value.
`n<`***text***`>`	***Negative/Right.*** Insert ***text*** to the right of the representation of a negative value.
`p<`***text***`>`	***Positive/Left.*** Insert ***text*** to the left of the representation of a non-negative value.
`q<`***text***`>`	***Positive/Right.*** Insert ***text*** to the right of the representation of a non-negative value.
`r<`***text***`>`	***Background.*** First fill the field by cyclically repeating ***text*** across it. Then overwrite the digits and decorators needed to represent the value. Leave the background text visible at positions that are not used for digits, sign, or decorators.
`s<`***text***`>`	***Symbol substitution.*** In each pair of characters within ***text***, the first indicates a character used by default and the second the desired substitute. For example, `s<, .,>` displays blank where `⎕fmt` would otherwise put comma, and comma where `⎕fmt` would otherwise put dot.

Figure 16–7: Summary of `⎕fmt` *decorators.*

< >	Text started with the left member of one of these pairs continues until the matching right member of the pair.
≤ ≥	
⊂ ⊃	
⁚⁚ ⁚⁚	
/ /	
⍞ ⍞	
⍕ ⍕	

Figure 16–8: Text-delimiting characters for decorator text.

Examples of ⎕*fmt*

```
      n
4163645361 0 ¯4163645361 3645361 14163645361

      'bcm<->k-2f14.2' ⎕fmt n
 41,636,453.61

-41,636,453.61
     36,453.61
141,636,453.61


      'm<(>n<)>q< >bck-2f17.2' ⎕fmt n
   41,636,453.61

  (41,636,453.61)
       36,453.61
  141,636,453.61

      '<Phone >,g<z (zzz) 999-9999>' ⎕fmt n
Phone   416)  364-5361
Phone        000-0000
Phone  ¯416)  364-5361
Phone        364-5361
Phone 1 (416) 364-5361
```

Obsolescence of "Paren/Semicolon" Argument to ⎕*fmt*

When ⎕*fmt* was first introduced, APL lacked a way of including character and numeric data in a single array. ⎕*fmt* therefore permitted ⍵ to be written as a series of separate expressions set off from each other by semicolons, and the whole enclosed in parentheses. This dodge is no longer required. Arrays of different type can be formatted together by an expression such as:

```
f ⎕fmt a⊃b⊃c
```

The *ad hoc* solution involving parens and semicolons is regarded as obsolescent because it is inconsistent with the rest of APL syntax.

SHARP APL continues to permit expressions such as

```
f ⎕fmt (a;b;c)
```

but this style is not recommended.

⎕*fx*

Fix a Definition

Monad ⎕*fx* *fixes* (that is, makes available for execution) a user-defined verb from the characters of its *canonical representation.*

⍵ is a character table containing the verb's definition in the form produced by ⎕*cr*. Each row of the table represents a line of the definition, without leading or closing ∇ and without line numbers.

Side effect: The verb defined by ⍵ is *fixed* (that is, materialized in the workspace). A definition can be fixed provided the name shown in ⍵ is available; that is, has no visible use, or its visible referent is to a verb (in which case the new definition replaces the old).

Result: The name of the verb thus fixed. When for any reason the definition cannot be carried out, the result is a numeric item identifying the line number[12] that contains the error. (Note that this is the same as an index into the rows of ⍵ when ⎕*io*←0. In 1-origin, line 1 of the proposed verb is located on row 2 of ⍵, since the first row of ⍵ is devoted to the header.) The result may indicate a row 1 greater than the index of the last row. That happens when ⍵ is empty, or the error is detected after the last row ⍵ has been processed.

12 The return code from ⎕*fx* differs from that returned by 3 ⎕*fd*, which reports the character index into its list argument rather than the line number.

When the name of the verb to be fixed has been localized, the newly defined verb is local. It ceases to exist when the verb to which it is local completes execution.

`3 ⎕fd` provides similar capabilities.

⎕hold
Hold Tied Files

See `⎕fhold`, earlier in this chapter.

⎕ht
Horizontal Tab

For ASCII terminals, the system noun `⎕ht` is used to describe the positions of tab stops in the terminal, so that the SHARP APL session handler can interpret *tab* characters when they occur in input, or make use of them in transmitting output. The value of `⎕ht` has no effect in a batch task, nor on output from an S-task; however, it is used by AP1 to interpret tab characters in the input.

At sign-on, `⎕ht` is an empty list. You may set it as a monotonically increasing numeric list, indicating the displacement of the stops from the left edge of the display, or as an item, indicating the interval at which tabs are set. When the input processor encounters a *tab* character coming from the terminal, it replaces it by the number of blanks needed to reach the next tab position. This replacement is done before the characters reach the APL workspace. There is no *tab* character in the APL character set.

`⎕ht` is a *session noun* (that is, one whose visible value survives `)clear` or `)load`. At each load or clear, the visible value in the old active workspace is copied to become the visible value in the new active workspace.

⎕io
Index Origin

The noun `⎕io` establishes the index origin used by several APL primitives. You may set `⎕io`, but the only acceptable values are an item `1` or `0`. In a clear workspace, `⎕io` is initially set to `1`.

When generating or using index values, SHARP APL assumes that they start with `⎕io`. Compare the effects produced by leaving `⎕io` at its default value, or setting it to the optional value `0`:

```
      ⊢⎕io←1
1

      ⍳5
1 2 3 4 5

      ⊢x←5+⍳5
6 7 8 9 10

      x[3]
8

      x[5]
10

      x[0]
index error
      x[0]
       ^
      3?3
3 1 2

      v←6 23 11 4 ¯6
      ⍋v
5 4 1 3 2

      x,[0.5] v
  6  7  8  9  10
  6 23 11  4  ¯6
```

```
      ⊢⎕io←0
0

      ⍳5
0 1 2 3 4

      ⊢x←5+⍳5
5 6 7 8 9

      x[3]
8

      x[5]
index error
      x[5]
       ^

      x[0]
5

      3?3
2 0 1

      v←6 23 11 4 ¯6
      ⍋v
4 3 0 2 1

      x,[0.5] v
5   6
6  23
7  11
8   4
9  ¯6
```

Verbs Affected by ⎕*io*

The value of ⎕*io* is used in connection with:

⍳	Both monad (to generate consecutive integers) and dyad (to return the indexes of ω in α).
?	Both monad (roll) and dyad (deal) to generate random integers.
⍋ ⍒	Both monad (numeric) and dyad (character) grade up and grade down.
⍉	Left argument of transpose.

`[ ... ]` Selection by indexing. Indexing to select the axis of application of ⌽ ⊖ , / \.

□lc
Line Counter

For each active user-defined verb, the system noun □lc shows the number of the line on which SHARP APL is currently working or (when execution is halted) is next to be executed when work resumes.

For example, if at the moment SHARP APL is working on line 3 of verb `report` and `report` was invoked by line 5 of `analyze`, the value of □lc is `3 5`.

If you signal a *weak interrupt* while line 3 is active, SHARP APL completes execution of the line and halts. The value of □lc is then `4 5`. But if you signal a *strong interrupt* SHARP APL halts at once, perhaps without completing execution of line 3, and in that case □lc is still `3 5`.

□lc is set by SHARP APL; you can use it but not set it. The numbers in □lc are the numeric part of the *state indicator* (which shows both the name of each active verb and the current line number). See also `2 □ws 2` later in this chapter, and `)si` in Chapter 17, "System Commands".

Like the state indicator itself, □lc includes an entry not only for each active user-defined verb, but also for each pending use of ⍎ or □-input. The value reported for ⍎ or □ is the *line number* of the line whose sentence invoked ⍎ or □ (usually, the value that appears next in the result of □lc). For example, if line 6 of `analyze` contains the phrase `⍎'x tri y'` and you interrupt execution before line 2 of `tri` has been completed, the state indicator looks like this:

```
      )si
tri[2]*
⍎
analyze[6]
```

And the value of □lc is `2 6 6`.

When a user-defined verb has been interrupted, the expression →□lc is commonly typed from the keyboard to mean "Resume execution with the next line in sequence." This is a shorthand for →□lc[□io], since → considers *only the first* of the values in its argument.

⎕lib
File Library

ω is a numeric item (or a one-item list) containing a single library number, or a two-element list in which the first item is a library number, and the second is a library passnumber. For example

```
⎕lib 1↑⎕ai
```

returns the names of files in your own private library.

When ω indicates a private library other than your own, the result includes a file only when its owner has set the file's access table in a way that grants you some sort of access to it, or when the library access table grants you appropriate access to view that library.

File permission: Any.

Library Permission: `4` to list file names that you have some access to in the specified library. `12` to list `all` file names in the specified library.

Result: A character table, with one row per file. It includes all files in the indicated library to which you have some form of access. Each row is a 22-element list, in which the first 10 characters contain the library number, right justified with leading blanks. The next character is blank. The last 11 characters contain the file name, left justified, with trailing blanks. The rows are not sorted; they are returned in fortuitous order. See also Chapter 14, "APL Component Files".

⎕load
Load a Workspace
⎕qload
Load a Workspace Quietly

Monads ⎕load and ⎕qload have the same effect as the system command)load. Each replaces the present contents of the active workspace with the contents of a saved workspace.

The argument is a character list (or item) identifying the workspace to be loaded (optionally preceded by the number of the library to which the workspace belongs). When you do not specify a library, your own is assumed.

It is a moot question whether ⎕load or ⎕qload returns a result: when SHARP APL has executed ⎕load, the active workspace in which ⎕load was executed no longer exists.

The two forms `⎕load` and `⎕qload` differ only in whether they display the message `saved` followed by the date and time at which the saved workspace was saved. `⎕load` causes the display, `⎕qload` (`q` for "quiet") does not.

Following either `⎕load` or `⎕qload`, SHARP APL automatically executes the line it finds stored as the visible value of `⎕lx` in the workspace just loaded. Applications that make use of `⎕load` require some mechanism to pass control from the workspace that invoked `⎕load` to the workspace loaded in consequence. Usually the invoking workspace depends upon the `⎕lx` of the new workspace to initiate action. Information may be passed between the two workspaces by way of `⎕sp`, or by a convention that the new workspace will look in a particular file for information previously stored there by the preceding workspace.

⎕lx

Latent Expression

The system noun `⎕lx` is one you may set before saving a workspace in order to control the way the workspace starts each time it is loaded. At each load, SHARP APL automatically executes the line it finds stored as the visible value of `⎕lx`, it in the same way that it would execute a line entered from the keyboard. `⎕lx` thus provides a mechanism for making an immediate and automatic start on such things as conditioning the workspace environment, starting or restarting an application, or validating a user's access to an application.

A valid `⎕lx` must be a *character item or list.* In a clear workspace, the initial value of `⎕lx` is `''`.

Whenever you load a workspace (*except* when you load it by using the `)xload` command) SHARP APL executes `⍎⎕lx`. If a sentence stored in `⎕lx` is invalid, SHARP APL reports an error and suspends execution in the same way that it would for a sentence entered in immediate execution.

Example of ⎕lx

A technique for passing information between workspaces is to store the information needed for continuity in the session noun `⎕sp`, so that the transition between workspaces looks like this:

In workspace *ws1*:

In verb *work1*:

```
      ...
[ ]   ⎕sp←⎕pack 'x y z'
[ ]   ⎕qload 'ws2'
```

In workspace *ws2*:

```
      ⎕lx←'work2'  ⍝Executed automatically when loaded
```

In verb *work2*:

```
...
[ ]  →(test ⎕sp)/0          ⍝Quit if ⎕sp not as expected
[ ]   ⎕pdef ⎕sp
[ ]    Make use of x  y  z etc.
[ ]   ...
```

The preceding example depends on the two workspaces following each other during the same session, so that ⎕*sp* can be used to link them. The following example illustrates a similar use of ⎕*lx*, but one that depends instead on a file (which might have been created earlier than the current session).

```
    ∇ work2;f
[1]  ⍝ When file 'command' exists and is tied to tn
[2]  ⍝ execute the sentence from its component 2
[4]  →('command     '≡11↓,(⎕nums=tn)⌿⎕names)↓0
[5]   ⍎ ⎕read tn,2
    ∇
```

When *ws2* is loaded and its conditions are met, it starts executing the instructions left for it in the file called *command*. However, if *work2* finds the file *command* is not tied, or is tied to some other tie number, it exits without going further. This technique allows a utility workspace in a library to be used automatically for different purposes by different users.

□names
Names of Tied Files

The system noun □names provides a table containing the file names of files currently tied by this task. See also Chapter 14, "APL Component Files". The result contains no indication of passnumbers.

File permission: None needed.

Result: A character table of the file names currently tied by this task, one file per row. The format of the result is the same as the result of □lib. Provided you do not □create, □stie, □tie, □erase, or □untie a file between invoking □names and □nums, these two nouns have their values in the same order.

□nc
Name Class of Identifiers

Monad □nc returns the *name class* of each of the names in ω. The argument is a table with one name per row. The result is a numeric list of classification codes, describing the visible (that is, most local) referent of each name thus:

Value	*Class*
0	Not in use
1	Label
2	Noun
3	Verb
4	Other

A value of four indicates that the identifier is a distinguished name (beginning with a quad symbol) or a group, or that the argument is not a well-formed name.

□nl
Name List

Monad □nl returns a table of names in the workspace whose visible (that is, most local) use is in a particular class. ω is a numeric item or list indicating the class of interest.

The classes that appear in the right argument are identified by the numbers 1 through 3, with the same meanings as in the result of □nc, above. For example, □nl 1 2 returns a list of names whose visible use is as a label or a noun.

The result is a table, with one name per row, arranged in alphabetical order (sorted so that all first-alphabet characters occur before any of the second-alphabet characters).

Dyad `⎕nl` restricts the result to names whose first letter is a member of α. For example,

```
'XyZ' ⎕nl 2
```

returns a table of the nouns that begin with *X*, *y* or *Z*.

⎕nums
Tie Numbers of Tied Files

The system noun `⎕nums` provides a list containing the tie numbers of files currently tied by this task. The order of the items in `⎕nums` matches the order of the rows in `⎕names`. See also Chapter 14, "APL Component Files". The result contains no indication of passnumbers.

File permission: None needed.

Result: A list of the tie numbers of files currently tied by this task. Provided you do not `⎕create`, `⎕stie`, `⎕tie`, `⎕erase`, or `⎕untie` a file between invoking `⎕names` and `⎕nums`, these two nouns have their values in the same order.

⎕out
Record Session I/O to File

Monad `⎕out` permits you to control the destination of the APL session's display. You can divert to a file what would otherwise appear at the terminal (of a T-task) or at the controlling shared name (of an S-task). Or you can both capture the output in a file *and* display it in the usual way.

Use of `⎕out` also permits you to capture normal output generated by a batch task, which would otherwise be lost.

ω is an integer list of either zero or two elements. When ω is empty, `⎕out` reports the current setting. When ω is a two-element list, the value provided in ω takes effect, and the result is the former setting.

When ω is not empty, its elements have significance as follows:

`ω[1]` ***Display.*** Whether output is sent to the display device of a T-task or the controlling shared name of an S-task:

- `0` No display
- `1` Display

`⍵[2]` ***File.*** Whether a record of output is appended to a file.
`0` Do not file.
`tn` Append output to the file tied to *`tn`*.

When `⍵[2]` is not zero, the system appends the output data as components of the APL file tied to `⍵[2]`. (You must have tied the file, without a passnumber, beforehand.)

If SHARP APL finds it is unable to redirect output as requested (for example, because the file is not tied, or is tied but is full), it executes `⎕out 1 0` on your behalf, and signals the event in the usual way (for example, as *`file tie error`* or *`file full`*).

Timing of ⎕*out* Result

The change in output routing produced by `⎕out` takes effect before the result of `⎕out` is returned. Thus, if your program permits the result of `⎕out` itself to be displayed, so that your program includes sentences like this,

```
[ ]   ⎕out 0 45        ⍝Start sending output to file 45
[ ]   Output
[ ]   ⎕out 1 0         ⍝Resume sending output to terminal
```

you will find that the first component written to the file contains `1 0` (the result of your initial instruction `⎕out 0 45`, indicating what the condition was before you executed `⎕out 0 45`), and when your program executes `⎕out 1 0` to return output to the terminal, you will see its result `0 45` (the former setting) displayed at the terminal, but not in the file.

Control Messages

While copying output to a file, the system may also generate *control messages* to describe the circumstances of input or output. When you subsequently examine the contents of the file, the control messages have to be interpreted (rather than printed). The control messages generated by `⎕out` are a subset of those recognized by the SHARP APL facility called HSPRINT. The list that follows shows all the control messages recognized by HSPRINT; those that may arise in a file recorded by `⎕out` are marked by an asterisk.

A control message is a 22-character list whose positions have significance as follows:

1 The character `⎕av[⎕io+255]`.

2–5 Message number. four-digit number, with leading zeros. The numbers refer to the list of standard control messages (see below).

6 Blank.

7–10 Component count. four-digit number, with leading zeros. The number of following components to which this control message applies.

11 Blank.

12–22 Mnemonic. This field is ignored by HSPRINT. In output generated in response to □*out*, the system writes here a standard mnemonic to help human readers interpret the control message. The same mnemonics are generated by the utility routines in workspace `1 hsprint`.

List of Control Messages

`0000 000n format`	Arguments to □*fmt* follow. The following components are not the *result* of formatting but the arguments. Formatting is left to the program that interprets the □*out* file. Where the "number of components" field indicates more than two, those after the first arose from the paren/semicolon convention for the right argument of □*fmt*.
`0001 000n mixed`	Mixed output follows. The following components are not the *result* of formatting but the arguments.
`0002 0001 pagen`	New page number follows.
`0003 000n quadp`	⍞-output follows.
`0004 0001 translate`	Translate table follows. The table maps APL characters to the output device, and is specific to the device employed.
`0005 0001 digits`	New value for □*pp* follows. The new value affects the display of subsequent components containing arrays that were stored unformatted, for example for mixed output.
`0006 0001 width`	New value for □*pw* follows. The new value affects the display of subsequent components containing arrays that were stored unformatted, for example for mixed output.
`0007 0001 title`	Page title follows.
`0008 0001 subtitle`	Page subtitle follows.

0010 0001 *prtuctm*	Control for printing user control messages follows. When the following component contains 1, control messages are printed; when 0 (the default), they are not.
0011 0000 *page*	Skip to beginning of new page.
0012 0001 *carriage*	Vertical forms control follows. The control is a Boolean list with as many elements as lines on the output page, and 1 for each line on which printing is permitted.
0013 0001 *prtarbout*	Control for printing □*arbout* data follows. When the following component contains 1, arbitrary output is printed; when 0 (the default), it is not.
0014 000*n* *arbout*	□*arbout* argument follows.
0015 000*n* *arbin*	□*arbin* prompt and result follows.
0016 0001 *prtarbin*	Control for printing □*arbin* prompt and data follows. When the following component contains 1, the arbitrary prompt and resulting input are printed; when 0 (the default), they are not.
0018 000*n* *hfile*	Identifies a file created by HSPRINT under its print-to-file option.
0019 0001 *quad er*	□*er* follows.
0020 000*n* *input*	Normal input follows.
0021 000*n* *quad*	□-input follows.
0022 000*n* *quad prime*	⍞-input follows.
0023 000*n* *fn defn*	Function definition input follows.
0024 000*n* *ws full*	Workspace full while preparing input.
0025 000*n* *st full*	Symbol table full while preparing input.
0026 000*n* *defn err*	Definition error while preparing input.
0027 000*n* *not in defn*	Obsolete: Formerly "Input not allowed in definition mode."
0028 000*n* *open quote*	Obsolete: Formerly "Input contained unmatched quote."

`0029 0001 prtinput`	Control for printing input follows. When the following component contains `1`, input is printed; when `0` (the default), it is not.
`0030 0001 quadps`	New value for `⎕ps` follows.
`0031 0001 char error`	Obsolete: Formerly "Character error while preparing input."

⎕pack

Build a Package

The verb `⎕pack` builds a package. A *package* is a noun containing a set of *names*, a set of *name classes*, and the *referent* of each name. A referent may be a user-defined verb, a noun of any type (including a package), or undefined.

Dyad α `⎕pack` ω forms a one-member package whose referent is the noun or pronoun ω, to which is assigned the name indicated by the character list α. (A single-character name may be represented by an item.)

For monad `⎕pack`, ω is a *namelist*, that is, either a character item or list in which successive names are separated by blanks, or a table with one name per row. The result is a package containing the names of ω together with the visible referent of each name.

The representation of a verb within a package loses any function monitoring that may have been established by `⎕fm`, but retains the original verb's trace- and stop-controls.

⎕pdef

Package Define

⎕ppdef

Protective Package Define

`⎕pdef` defines objects from within a package. For each of the names in package ω, monad `⎕pdef` causes the referent within the package to replace the visible referent outside.

Dyad α `⎕pdef` ω does the same thing, but only for the names included in namelist α. The namelist may be a table with one name per row, or an item or list of names delimited by blanks.

The objects thus defined replace the visible (that is, most local) referent of those names. When the name of an object to be defined corresponds to a

name that has been localized in a verb that is currently executing, pendent, or suspended, the newly defined object is local, and ceases to exist when the verb to which its name is local completes execution.

Result: None for □*pdef*.

Protective Definition

The extra *p* in the name □*ppdef* stands for *protective*. □*ppdef* works in the same way as □*pdef*, both for monad and for dyad, *except* that a name is defined from within package ω only when it is not already in use outside. Thus existing uses of the names are "protected."

The result of □*ppdef* is a table containing the names that were *not* successfully defined; when definition is completely successful, the resulting table has no rows.

□*pex*

Package Expunge

The result of α □*pex* ω is a package containing a subset of the names in the package ω, together with their referents.

α is a *namelist* (that is, either a character list in which successive names are separated by blanks or a table with one name per row) containing names to be excluded from the result. It is not necessary that the names in α actually exist in ω.

□*pins*

Package Insert

Dyad □*pins* merges two packages. Both α and ω are packages. The result is a package containing the union of the sets of names in the argument packages.

Where the same name occurs in both α and ω, the result takes the referent from ω. Thus everything in ω appears in the result, whereas α contributes only those names and referents *not* contained in ω.

□plock

Package Lock

□plock takes a package argument and returns a package in which verb definitions are locked.

The result of monad □plock is a package similar to ⍵ in which *all* verb definitions are locked.

The result of dyad □plock is a package identical to ⍵, but in which verbs present in ⍵ *and* named in ⍺ are locked. ⍺ is a namelist (that is, a character list of names separated by blanks, or a table with one name per row).

In the result, all nouns, and verbs present in ⍵ but not named in ⍺ are unchanged.

□pnames

Package Names

Monad □pnames reports the names contained in package ⍵. The result is a table with one row for each name in ⍵. The ordering of names in the result is fortuitous.

When ⍵ is an empty package, the result is a 0-by-0 table. However, when ⍵ is not a package, the result is an empty list. The expression ⍴⍴□pnames n thus provides a test of whether a noun n is or is not a package.

□pnc

Package Name Class

□pnc reports the name class of names in a package. The result is a numeric list, one item per name, representing the class of each name by an encoding similar to that returned by □nc:

Value	*Class*
¯1	Undefined, but in package
0	Referent not in package
1	[unused]
2	Referent is noun
3	Referent is verb
4	[unused]

Monad □pnc returns the name class of every name in ⍵, in the same order as □pnames ⍵.

For dyad □*pnc*, α is a *namelist*, that is, either a character list in which successive names are separated by blanks, or a table with one name per row. The result contains the name class of each of the names in α, in the order that they appear in α.

□*pp*

Print Precision

The system noun □*pp* specifies the maximum number of significant digits, or print precision, provided by the system when it displays numeric floating-point data using either default output or ⍕.

You may set □*pp* as an integer item, not less than 1 nor greater than 18; in a clear workspace, the default value is 10.

□*pp* has no effect on the display of any value that can be represented internally as an integer, or on a number that is not different from an integer, even when represented internally in floating point. For example:

```
      □pp←3
      ÷3
0.123

      1234.0000000000001
1.23e3

      1234.0000000000000001
1234
```

When □*pp*←18, the result permits display of full internal precision, with every internal floating-point value distinguishable from its nearest neighbors. The final digit may not be otherwise significant.

□*ppdef*

Protective Package Define

See the discussion of □*pdef*, earlier in this chapter.

□ps

Position and Spacing in Display

The system noun □ps governs the formatting of an array of boxed items.

In an array of rank 2 or greater, the alignment of rows and columns is preserved by padding a narrow item to match the width of the widest item in its column, or padding a short item to match the tallest item in its row. When the display of an item requires less space than reserved for it, the first two elements of □ps determine its position within the rectangle assigned to it.

The second two items control the amount of additional space inserted vertically or horizontally to separate adjacent items. (These apply to any array, not just those of rank 2 or greater.) A negative value causes a box-drawing character to be placed at the vertical or horizontal edges of each box.

The effects of the four elements of □ps are as follows:

□ps[1] ***Position in grid, first axis (vertical).***

- ¯1 At the top of the available space.
- 0 Centered in the available space.
- 1 At the bottom of the available space.

□ps[2] ***Position in grid, second axis (horizontal).***

- ¯1 At the left of the available space.
- 0 Centered in the available space.
- 1 At the right of the available space.

□ps[3] ***Vertical separation.***

Magnitude: the number of additional rows interpolated vertically in the result between the representations of successive rows of ω.

Vertical box boundaries.

When □ps[3] is negative and has a magnitude of at least 2, the top and bottom edges of a box are marked by ¯ at the top of the box and _ at the bottom. Where necessary, these box-edge characters overwrite the interpolated blank rows (which is why there must be at least two of them). Where necessary, the result contains extra rows at the top or bottom to accommodate box boundaries that would otherwise project beyond the outer edges of the result.

□ps[4] ***Horizontal separation.***

Magnitude: the number of additional columns interpolated horizontally in the result between the representations of successive columns of ω.

Horizontal box boundaries.
When `⎕ps[4]` is negative and has a magnitude of at least 2, the left and right edges of a box are marked by | at each side. Where necessary, these box-edge characters overwrite the interpolated blank columns (which is why there must be at least two of them). Where necessary, the result contains extra columns at the left or right edges to accommodate box boundaries that would otherwise project beyond the outer edges of the result.

⎕psel

Package Select

The result of dyad `⎕psel` is a package containing a subset of the names in the package ω, together with their referents.

α is a *namelist*, that is, either a character list in which successive names are separated by blanks, or a table with one name per row. α specifies the names to be selected from package ω (and which therefore appear in the result).

Each name in α must actually occur in ω (although it may exist there without a referent). SHARP APL rejects an expression in which a name in α does not exist in ω as a *domain error*. Monad `⎕psel` is undefined.

⎕pval

Package Value

Dyad `⎕pval` extracts the value of a noun from package ω. α is a character list (or item) containing a single name. The name's referent in ω must exist and must be a noun.

The result is the value of the noun α from in package ω. For example:

```
      a←1 2 3
      b←'Tom'⊃'Dick'⊃'Harry'
      p←⎕pack 'a b'
      2×'a' ⎕pval p
2 4 6
```

□pw

Page Width

The system noun □pw sets the maximum length of a line displayed by the session handler in response to default output, or output produced by □- or ⍞-output.[13] A line broken because it would exceed □pw is said to be *folded.* After a fold, each continuation segment is indented by six positions. For further discussion, see the section "Display of Nouns" in Chapter 4, "Nouns and Pronouns", and the discussion of ⍕ in Chapter 6, "Verbs".

You can set □pw as an integer item not less than 30 nor greater than 250. In a clear workspace, the default value of □pw is usually 132. However, this value is site-dependent, so it may be different at your site. In SHARP APL/PC, the value is 80. The current value of □pw is included when you save a workspace, and reactivated with the rest of the workspace when you subsequently load it.

□qload

Load a Workspace Quietly

See the description of □load, earlier in this chapter.

□rdac

Read File Access Table

Monad: □rdac ***tn*** **[,*pn*]**

Read file access table of file tied to ***tn***, with optional passnumber ***pn***. See the discussion of file access controls in Chapter 14, "APL Component Files".

where ***tn*** is a tie number and ***pn*** the passnumber (required only when you tied the file using a passnumber).

File Permission: 256 or 4096.

Result: The three-column integer library access table of a tied file. The table contains the following information:

(□rdac ⍵)[1] User number to which access is granted. 0 means "anyone."

(□rdac ⍵)[2] Access level granted to the user numbers identified in column 1. The level is encoded as the sum

[13] Output produced by □arbin or □arbout is not constrained by □pw and may disrupt the alignment of ordinary output if they are intermixed. See the discussion of □arbin and □arbout earlier in this chapter.

of each of the permissions granted. Since each permission is a power of 2, the sum can represent any access to be granted or denied independently of the others. Access `¯1` grants complete access. See the discussion of access controls in Chapter 14, "APL Component Files".

`(⎕rdac ⍵)[3]` Passnumber required to tie the file (and at each use of a file verb while the file is tied). `0` in this column means "No passnumber required." See the discussion of passnumbers in Chapter 14, "APL Component Files".

⎕rdacl
Read Library Access Table

Monad: `⎕rdacl` ***lib*** **[*,lpn*]**

Monad `⎕rdacl` reads the library access table of library ***lib***, using optional library passnumber ***lpn***. The argument is a library number, followed (when required) by a library passnumber).

An access table consists of three columns, in which the first is a user number, the second a *permission code*, and the last a passnumber. The significance of the access table is discussed in the section "Library Access Controls" in Chapter 14, "APL Component Files".

Library Permission: `16`.

Result: The library access table of a specified library. The table contains integers in three columns, as follows:

`(⎕rdacl ⍵)[1]` User number to which access is granted. `0` means "anyone."

`(⎕rdacl ⍵)[2]` Access types granted to the user numbers identified in column 1. The level is encoded as the sum of each of the permissions granted. Since each permission is a power of 2, the sum can represent any access to be granted or denied independently of the others. Access `¯1` grants complete access. See the section "Library Access Controls" in Chapter 14, "APL Component Files".

`(⎕rdacl ⍵)[3]` Passnumber required to obtain the specified access. A `0` in this column means "No passnumber required." See the section "Library Access Controls" in Chapter 14, "APL Component Files".

⎕rdci
Read File Component Information

Monad: ⎕rdci ***tn, cn*** **[,*pn*]**

Read component information for component ***cn*** of the file tied to number ***tn*** with optional passnumber ***pn***.

The argument identifies a particular component in a particular file. The argument is a list of either two or three integers. The first contains the tie number of a tied file. The second is the number of a particular component in that file. If you specified a non-zero passnumber to tie the file, there must also be a third element containing that passnumber.

File Permission: 512.

Result: a three-element numeric list containing the following information about the indicated file component:

- `(⎕rdci ⍵)[1]` Size of the file component in bytes. The size reported includes both the data itself and the internal data description, which specifies its type, rank, and shape.
- `(⎕rdci ⍵)[2]` User number of the task that wrote the component's current value.
- `(⎕rdci ⍵)[3]` Timestamp showing when the component's current value was written. Time is recorded in 60ths of a second since 1960-03-01 at 00:00:00.

The representation in 60ths may be converted to days, hours, minutes and seconds by the expression:

```
0 24 60 60 60⊥(⎕rdci ⍵)[3]
```

The verb *ftt* in workspace `1 ts` will convert the timestamp into ⎕ts format. The number of days may be converted to a Julian day number by adding 2436995 (the day number of 1960-03-01), and then to a conventional date by a definition such as the following (by E.E. McDonnell):

```
    ∇ ymd←daterep J;y;d;m;c;q;r;⎕ct
 [1]  ⎕ct←2*¯44                  ⍝Set default; fails if ⎕ct=0
 [2]  J←J-1721119                ⍝Remove astronomer's constant
 [3]  c←⌊(J-0.25)÷36524.25       ⍝Completed centuries
 [4]  d←⌊¯0.25+J-c×36524.25      ⍝Days after centuries
 [5]  q←⌊(d+0.75)÷365.25         ⍝Find quarter
 [6]  d←⌊1.75+d-q×365.25         ⍝Year starts March 1
 [7]  m←⌊(d-0.6)÷30.6            ⍝Completed months
 [8]  d←⌊0.4+d-m×30.6            ⍝Days in remaining month
 [9]  y←(c×100)+q+m≥10           ⍝Adjust year to start Jan 1
[10]  m←1+12|m+2                 ⍝And adjust month number
[11]  ymd←y,m,d
    ∇
```

Verbs such as `time` and `timen` in workspace `1 fileaid` may also be of utility.

⎕rdfi

Read File Timestamp Information

Monad: ⎕*rdfi* ***tn*** **[*,pn*]**

where ***tn*** is a tie number and ***pn*** the passnumber (required only when you tied the file using a passnumber).

Result: A numeric table of shape `4 2`, containing user number and timestamp information about the file. The number of rows or columns in the result may change in future releases of SHARP APL. Do not make assumptions about the shape of the result of ⎕*rdfi*.

`(⎕rdfi ⍵)[1;]` The first row of information relates to the creation of the file using the ⎕*create* system verb. It contains the user number that created the file, followed by the timestamp of when the file was created.

`(⎕rdfi ⍵)[2;]` The second row relates to the latest setting of the file access table using the ⎕*stac* system verb. It contains the user number that last set the access table, followed by the timestamp of when the access table was set.

`(⎕rdfi ⍵)[3;]` The third row is reserved for future use, and is currently set to `¯1 ¯1`.

`(⎕rdfi ⍵)[4;]` The fourth row relates to the latest alteration of the file, using the system verbs ⎕*append*, ⎕*appendr*, ⎕*drop*, ⎕*rename*, ⎕*replace*, or ⎕*resize*. It contains the user number that last altered

the file, followed by the timestamp associated with that alteration. This information might be used to implement a file caching scheme for shared files.

The timestamps are recorded in 60ths of a second since 1960-03-01 at 00:00:00. The representation in 60ths may be converted to days, hours, minutes, and seconds by the expression given in the description of `□rdci`, above. The verb `ftt` in workspace `1 ts` will convert the timestamp into `□ts` format.

An example of the use of `□rdfi` is as follows:

```
      □rdfi tn
 1.726354000e6   5.580219021e10
 1.726354000e6   5.589528844e10
¯1.000000000e0  ¯1.000000000e0
 7.858878000e6   5.597212603e10
```

File Permission: `65536`.

See also Chapter 14, "APL Component Files".

□read

Read Component of a Tied File

Monad: `□read` ***tn, cn*** **[,*pn*]**

Read data object in component ***cn*** of file tied to ***tn***, with optional passnumber ***pn***.

The right argument identifies a single component of a tied file. It is a list of two or three integers.

The first element contains the tie number of a tied file.

The second element contains the number of an existing component within that file. (If a component exists, its component number is less than the second element of the result of `□size`, but not less than the first.) The system rejects an attempt to refer to a non-existent file component with the message `file index error`.

The third element, when required, is the passnumber.

See also Chapter 14, "APL Component Files".

File permission: `1`.

Result: The value of the noun stored in the file component.

□rename
Rename a Tied File

Dyad: '[*library*] *newname* [*newres*] ' □rename *tn* [,*pn* [,*lpn*]]

Rename the file tied to *tn* so its name becomes *library newname.*

where *tn* is a tie number, *pn* the file passnumber (required only when you tied the file using a passnumber), and *lpn* the library passnumber. If the file library access table requires it, you may have to include a non-zero library passnumber *lpn* to perform the rename. Both passnumbers are optional, but the file passnumber must be specified if the library passnumber is required. See Chapter 14, "APL Component Files" for a discussion of passnumbers. The file's reservation may be changed by specifying a new reservation as *newres.*

newname is a character list containing the new name proposed for the file; it must be a well-formed name, not currently in use the name of a file in library *library*. An ill-formed name is rejected as a `domain error`.

The new file name may be preceded by a library number, *library*. If you omit the library number, your own library is assumed. You can change the name so that the file is moved from the library in which it first resided to your own library, or to a library to which you have access, provided that the file's access table includes permission for you to rename it, and the library access table includes permission to rename into that library.

It is not usually possible to transfer a file to someone else's private library with □rename, unless they have specifically given you permission to do so by including your user number in their library access table. For infrequent use, if you want to give a file to someone else, first give that person permission to □rename the file, and then ask that person to rename it (so that □rename is used by the recipient rather than the donor).

You can change the library to which a file belongs provided:

- the new library is your own or a public library
- the file's access table grants you permission to do so
- the source and target libraries are in the same volume class.[14]

File Permission: `128`

Library Permission: `2`. To invoke □rename with a file in a library other than your own, the file's access table must include rename permission *and*

[14] The assignment of library numbers to volume classes is done by the system administrators. In some installations all libraries may be in a single volume class, so this presents no obstacle. Other installations may use many volume classes and thus (indirectly) restrict the ability to rename a file to a different library.

the library's access table must include rename permission. See also the section "Library Access Controls" in Chapter 14, "APL Component Files".

Effect: The file is renamed as indicated. It now belongs either to your file library, or to the file library you indicated in the left argument. If the new library number is less than `1000`, you are the owner of the file. Otherwise, the library is the owner.

Result: None.

□replace

Replace Noun Stored in a Component

Dyad: α `□replace` ***tn, cn*** `[,`***pn***`]`

Replace component ***cn*** of file tied to ***tn***, with optional passnumber ***pn***, with noun α.

The right argument identifies the file tie and component number, and passnumber if required. The left argument is the noun whose value is to replace the value previously stored in that component. The right argument is a list of two or three integers.

The first element contains the tie number of a tied file.

The second element contains the number of an existing component within that file; see the discussion of the second element of the argument of `□read`.

The third element, when required, is the passnumber.

The left argument is a noun. It may have any size, shape, or type. There is no requirement that it have the same shape or type as the noun it replaces.

Effect: The value of α replaces the noun stored in the component. There is no requirement that the new noun match the former value in type, rank, shape, or size.[15]

For the system to make the replacement, *one* of the following must be true:

- The storage already in use for the file does not exceed the file's size limit
- The new component requires no more space than the old one.

Following a successful replacement, the component contains the data itself, together with a description indicating the type, rank, and shape of the data. It also contains control information indicating the number of bytes required

15 When the replacement requires more space than the former value, the file system in fact allocates entirely new space for the replacement and marks the old space as "garbage." The old space is not recovered until the system administrators run the periodic utility programs that dump and restore the files. Careless use of growing `□replace` can cause files to grow to very large sizes.

to store the object, the user number of the task that did the replacement, and a timestamp indicating the date and time at which the replacement took place; see the description of `□rdci`.

See also the discussion in Chapter 14, "APL Component Files".

File permission: `16`.

Result: None.

□resize
Set File Size Limit

Dyad: `α □resize` ***tn*** `[`***,pn***`]`

where ***tn*** is a tie number and ***pn*** the passnumber (required only when you tied the file using a passnumber).

α is the desired new file size limit, as a numeric item. Its value may not be less than the present actual size of the file, nor so large that the sum of the size limits for all files you own exceeds the file reservation set by the system administrators.

Effect: The file's size limit is revised as indicated, to the next larger multiple of physical record size. As long as the actual size of the file is not greater than the limit, the file system accepts a new component or a replacement for an exiting component. Once the file's actual size exceeds the limit, the file system rejects an attempt to append, or to replace a component with a larger one, with the message `file full`.

See also the discussion in Chapter 14, "APL Component Files".

File Permission: `1024`.

Result: None.

⎕rl
Random Link

The system noun ⎕rl is the seed value (or random link) used by the pseudo-random number generator. It is used in evaluating monadic ? (*roll*) and dyad ? (*deal*). Each time you invoke ?, as a side effect the system sets ⎕rl. Each time you invoke ? with the same argument and the same value of ⎕rl (and also of ⎕io), you get the same result. When you save a workspace, the value of ⎕rl is saved as part of it. Each time that saved workspace is loaded, the same uses of ? yields the same results.

You may set ⎕rl before executing the random function ? to assure:

- *Same* results as on some other occasion (for example, to reproduce a test conducted earlier). For repeatable results, set ⎕rl to whatever value it had before.
- *Different* results on each occasion (for example, when you repeatedly load a workspace but want to generate fresh random numbers each time). For unpredictable results, set ⎕rl to something unpredictable; for example:

```
⎕rl←(+/⎕ts)+×/3↓⎕ts+1
```

Range and Default: You can set ⎕rl to any integer item greater than 0 and less than 2^{31}-1. In a clear workspace, the default value is 16807 (which is 7^5).

Algorithm: As each pseudo-random number is generated, the seed (⎕rl) is first assigned a new value and immediately used in the computation. The new value of ⎕rl depends only on the previous value, and not on the arguments of ?. The visible value of ⎕rl is set as:

```
⎕rl←p|m×⎕rl ⊣ p←2147483647 ⊣ m←16807
```

This process is cyclic, and repeats after p–1 repetitions. See also the discussion of ? in Chapter 6, "Verbs". For more information on the linear congruential algorithm used, see *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, D.E. Knuth, pages 9–10.

□*run*
Start N-Task

Monad □*run* starts an N-task. Its result is a two-element numeric list containing the number of the task thus started (when the request was successful) or a return code indicating the nature of the trouble (when it was not). For a general description of APL tasks, see Chapter 2, "Tasks, Users, Libraries". For a discussion of N- and B-tasks, see Chapter 13, "Batch Tasks".

ω is a character list consisting of five fields separated by blanks. For example, an N-task might be launched by an expression such as the following. The division into fields is marked by underlines, and is discussed in the following description of the fields.

```
□run ': 501 Analysis NCrash 40 600'
      _ ___________ ______ __ ___
```

Owner Identifies the user number that will own the new task. Ordinarily, the owner is the same as the user whose task executed □*run*. In that case, a colon suffices (as illustrated). When the owner's user number is to be different, this field contains the user number, a colon, and the user's current sign-on password (but not the right paren that accompanies sign-on to a T-task or S-task). Either way, the field must contain a colon.

Initial workspace An N-task may start in either of two ways. This field determines which.

For autostart: The name of a workspace. The N-task will start by loading that workspace. (It is up to the workspace's latent expression to take things from there.) Note that there may be a blank between the library number and the name of the workspace (so that blank is permissible inside this field).

For split workspace: A colon. The N-task will start with a clone of the present active workspace. (It is up to the program now being executed to distinguish the parent workspace from the new N-task.)

Save-name Name to be assigned to the N-task's active workspace when it is saved at termination of the N-task. This workspace is presumed to be in a library of the N-task's owner. This field may not be elided.

MaxCPU The maximum number of CPU units for this task; 0 means no limit. When SHARP APL notices that the task has exceeded

its CPU limit, it terminates the task and saves its active workspace under the name indicated.

Max elapsed Maximum elapsed seconds for the task; `0` means no limit. When SHARP APL notices that the task has exceeded its elapsed time limit, it terminates the task and saves its active workspace under the name indicated.

Result Codes

The two-element numeric list returned by `⎕run` is interpreted as follows:

`0 0` New task. This is the value `⎕run` returns for an N-task started by cloning the parent task's active workspace.

`0 n` Task ***n*** has been successfully started. This is the value `⎕run` returns to the parent task.

`1 n` ⍵ contains an invalid character at position n.

`2 0` The SHARP APL system's capacity for N-tasks is exhausted.

`3 0` The proposed N-task would exceed the owner's task quota.

`4 0` The owner has already saved the full quota of saved workspaces, and hence could not save a workspace at termination of the proposed N-task.

`5 0` Attempt to load the workspace identified by ***Initial workspace*** produces `ws full`.

`6 0` Attempt to load the workspace identified by ***Initial workspace*** produces `symbol table full`.

`10 0` The proposed sign-on number or password is not recognized.

`11 0` The proposed sign-on number has been locked out.

`12 0` The proposed ***Initial workspace*** was not found.

`13 0` The ***Initial workspace*** did not have an executable `⎕lx`.

`14 0` The system's capacity for tasks is exhausted.

`15 0` The user is not authorized to use `⎕run`.

`16 0` Hardware error.

`17 0` Attempted to `⎕run` onto a constrained user number.

The following are applicable only to tasks started by the Batch Task Scheduler:

`58 0` `□sp` is too large.

`59 0` The system's crash recovery program was unable to recover the active workspace of an interrupted task.

`60 0` The Scheduler has been unable to start the task that, according to the B-task specification NBBTASK, must precede the proposed task.

`61 0` The Scheduler has been unable to complete the sort that, according the B-task specification NBSORT, must precede the proposed task.

`62 0` The task is not restartable; *ABNORMRST* was not specified.

`63 0` Error in processing specifications.

□*runs*

Report Running Tasks

This system noun reports the APL tasks running on your user number. The system sets it; you may refer to it but not set it.

□*runs* is a numeric table with one row for each task; the task that executed □*runs* is always in the top row. The table has eight columns, as follows:

□*runs*`[;1]` The task number.

□*runs*`[;2]` User number of the task's owner.

□*runs*`[;3]` User number of the tasks's starter.

□*runs*`[;4]` Type of task, as follows:

- `0` Interactive (T- or S-task)
- `1` N-task
- `2` B-task

□*runs*`[;5]` CPU units since start of the task.

□*runs*`[;6]` Elapsed seconds since start of the task.

□*runs*`[;7]` Task's CPU limit. This is `0` (no limit) unless set differently at the start of an N- or B-task.

□*runs*`[;8]` Task's elapsed time limit. This is `0` (no limit) unless set differently at the start of an N- or B-task.

Columns 2 and 3 may differ when you start an S-task by signing on with a different user number. (You have to supply the other user's password.)

□SC

Shared Variable State Change

The system noun □SC is interlocked to detect a change in the state of any of the names you are sharing.

The default value of □SC in a clear workspace is 0. Once you have established a valid clone ID (either explicitly or by default), the Shared Variable Processor:

- interlocks □SC so that you cannot use its value until the Shared Variable Processor has set it
- sets □SC to 1 each time there is a shared name event
- sets □SC to 1 when the SHARP APL system is beginning to undergo shutdown for some reason.

Thus, whenever you are able to use the value of □SC, you are assured that at least one shared name event has occurred since the last time you used □SC.

Setting a □SC Time-out

When you set the value of □SC, the Shared Variable Processor interprets the value you set as the number of seconds until a □SC time-out. The value you set must be a positive integer. If no shared name event has occurred by the end of that time, the Shared Variable Processor sets □SC to 1 anyway. Thus, when you are able to read the value of □SC you know either that there has been a state change or that the time you set has expired. See Chapter 11, "Shared Variables", for further discussion of state change.

Impending Shutdown

When SHARP APL system shutdown is imminent, the second element of (2 □WS 3)[□IO+1] is set to 1, instead of its normal value of 0. Using this information, tasks can perform internal housekeeping chores before the shutdown occurs (assuming that, for normal shutdowns, your site has a period of grace in which tasks can complete before they are interrupted.).

When an APL shutdown has been initiated, all tasks suspended for a state-change wait (a □SC-wait, not a shared name wait) are posted. This post can benefit applications that employ background tasks or server tasks. Consider the following generalized verb:

```
     ∇ serve
[1] go: Process                    ⍝ Work to do
[2]  →(2 □ws 3)[□io+1]⍴shut       ⍝ Is shutdown pending?
[3]  →(□sc=1)⍴go                  ⍝ Wait for some work.
[4] Shut: Cleanup                  ⍝ Clean up and terminate.
```

In the event of a shutdown, the *serve* verb does not immediately suspend on execution of □*sc*; it continues as if a post had just occurred. This gives programs the opportunity to detect and prepare for shutdown. This can streamline recovery procedures when SHARP APL restarts. For example, if the application writes a "clean shutdown" component to a file when it terminates normally, checking for this component would determine whether you need to perform the recovery and integrity checking procedures normally undertaken after abnormal termination.

□*signal*

Terminate and Signal Event

The verb □*signal* is conditional: what it does depends on whether or not its argument is empty. Therefore it is frequently used in a conditional expression such as:

```
□signal test/event
```

Given an *empty* ⍵, □*signal* does nothing.

Result: None.

When ⍵ is *not* empty, it must be a numeric item or one-element list. In that case, □*signal* causes an immediate exit from the verb highest on the execution stack (thereby cutting back the stack by one level).[16] In response to the signal, the interpreter sets the visible value of □*er* to record the event (just as it does from an event caused by an interrupt or an error in the definition). If a *trap* has been set for event ⍵, the trap's recovery expression takes over.

When the system sets □*er* following a signaled event, it inserts `4 0⍕⍵` in the first four positions of row 1. On the rest of row 1, it inserts the *event name*. The event name is:

For dyad □*signal* — The characters from ⍺.

For monad □*signal* — When ⍵ is one of the standard event numbers, the standard event name (subject to)*nlt*) associated with ⍵. Otherwise, blanks.

16 Notice that this is equally true when you type □*signal* from the keyboard, except that there is nothing to cut back if you type □*signal* while no verb is pending.

On row 2 it inserts the sentence that contains the verb whose execution was terminated by `⎕signal`. When that sentence is itself part of a user-defined verb, it shows the name of the verb and the number of the line that contains the sentence.

On row 3 it inserts a caret pointing to the name of the verb whose execution was terminated by `⎕signal`.

When there is *no* trap for event ω, execution halts. The system displays an error message in the usual way; that is, `⎕er` with the first four characters (the event number) omitted.

See the section on event trapping in Chapter 10, "Event Handling".

Result: None.

⎕size
Size of Tied File

Monad: `⎕size` ***tn*** `[,pn]`

Size of the file tied to ***tn***, with optional pass number ***pn***, returned as a the following four-element numeric list:

Result: This monadic verb returns a four-element numeric list describing the size of a file. The argument is the tie number of the file, followed (when required) by a passnumber. The elements of the list describe the file's size as follows:

`(⎕size ω)[1]`	Lowest component number for this file (that is, 1 for a file from which no leading components have been dropped).
`(⎕size ω)[2]`	Number of *next* component to be appended (that is, one more than the number of the last existing component).
`(⎕size ω)[3]`	Total space now occupied by the file, in bytes. This number includes the file's internal directories, and may include dead space when a file has grown by replacing some components with components of different size.
`(⎕size ω)[4]`	File size limit, in bytes. A request to `⎕append` or `⎕replace` with a larger record is honored only while the actual size is less than or equal to the limit.

If ω is the tie number of a newly created (and therefore empty) file having a size limit of 100992 bytes, then its size would be reported thus:

```
      ⎕size ω
1 1 0 100992
```

The number of components in the file tied to ⍵ is given by:

```
|-/2↑⎕size ⍵
```

See also Chapter 14, "APL Component Files".

File permission: `1` or `32768`.

⎕sp

Session Parameter

The system noun ⎕sp is propagated from one active workspace to another as your task performs)load, ⎕load,)clear, etc. It is often used to pass a starting argument to a new workspace. You may set it to any value you wish.

Warning: Setting ⎕sp to a value which occupies too much space may cause workspace full to occur when changing workspaces.

⎕stac

Set File Access Table

Dyad: ⍺ ⎕stac ***tn*** [***,pn***]

Set access table to ⍺ for the file tied to ***tn***, with optional passnumber ***pn***.

The right argument is the tie number of a file followed (when required) by a passnumber).

The left argument is the proposed access table. It is a three-column table of integers. SHARP APL requires that the argument have this form, and contain integers, but does not otherwise verify that the values in it are valid. The columns of the access table are respectively:

1 User numbers to which you grant access. 0 means "anyone."

2 File access level you grant to each user identified in column 1. The level is encoded as the sum of each of the permissions granted. Since each permission is a power of 2, the sum can represent any access to be granted or denied independently of the others. Access ¯1 grants complete access. See the section "File Access Controls" in Chapter 14, "APL Component Files".

3 Passnumber you require in order to tie the file (and at each use of a file verb while it is tied). 0 means "No

passnumber required." See the discussion of passnumbers in Chapter 14, "APL Component Files".

Effect: The left argument ⍺ becomes the access table for the file tied to ⍵. The system re-evaluates the permission of all tasks that currently have this file tied; your change to the access table takes effect immediately, even for tasks that had already tied the file.

The following gives to user `1154` unlimited access with no password, and to all users permission `3` (that is, actions `1` and `2`) provided they use passnumber `99`:

```
      mat←2 3⍴1154 ¯1 0 0 3 99
      mat
1154 ¯1  0
   0  3 99
      mat ⎕stac 4
```

Watch out: If you invoke `⎕stac` in such a way that you exclude yourself from `⎕stac` access to the file, you cannot do another `⎕stac` to restore it. Site policy *may* allow system administrators to invoke a facility called *filehelper* to assist you, if this occurs to a file that you own.

File Permission: `8192` or `256`.

Result: None.

⎕stacl
Set Library Access Table

Dyad: ⍺ `⎕stacl` ***lib*** **[**,***lpn***]

Set library access table to ⍺ for library ***lib***, using optional library passnumber ***lpn***.

The left argument is the proposed access table. It is a three-column table of integers. SHARP APL requires that the argument have this form, and contain integers, but does not otherwise verify that the values in it are valid. The columns of the access table are respectively:

1 User numbers to which you grant access to the specified library. `0` means "anyone."

2 Library access level you grant to each user identified in column 1. The level is encoded as the sum of each of the permissions granted. Since each permission is a power of 2, the sum can represent any access to be granted or denied independently of the others. Access `¯1` grants complete

access to the entire library. See the section "Library Access Controls" in Chapter 14, "APL Component Files".

3 Passnumber you require for the specified access. `0` means "No passnumber required." See the discussion of passnumbers in Chapter 14, "APL Component Files".

Effect: The left argument ⍺ becomes the access table for the specified library.

Library Permission: `32`.

Result: None.

□stie
Share Tie

Dyads `□stie` and `□tie` establish linkage between the name of an APL component file and the tie number used to refer to it from the active workspace. `□stie` permits other users to share the file at the same time (whereas `□tie` is exclusive).

'[*libnum*] *filename*' `□stie` *tn* [, *pn*]

The right argument is integer, containing the proposed tie number, followed (when required) by the passnumber called for in the file's access table.

The proposed tie number must be a *positive integer less than* 2^{31} and not in use by this task as a tie number of another file. Note that it does not matter what the tie number is, provided only that you choose a number that is not in use to tie some *other* file, and (as a practical matter) the applications referring to the tie number have a way to know what it is.

When the right argument has two elements, the second is the passnumber. When the argument is a scalar or one-element vector, a passnumber of `0` is understood.

The left argument is a character list (or item) containing the name of an *existing* file. If you omit indication of library, your own library is assumed.

The file named in ⍺ must be one that is not now exclusively tied by any task (including your own). An attempt to tie a file that another task has tied exclusively (or that your task has already tied in any way) is rejected as `file tied`.

Effect: The file is share-tied. You may proceed to use the file. Other tasks may have already share-tied the file, and may continue to do so. (See `□fhold` for mechanisms to coordinate concurrent use.) The tie number

(and passnumber, if you included one in ⍵ must appear in the argument of any of the file access verbs.

You are permitted to share-tie a file only if you are also permitted some sort of access to it. What that access is depends on the file's ownership and access table. When the file belongs to a user other than yourself, the file's access table specifies what file verbs you can use (see the discussion of file access in Chapter 14, "APL Component Files"). When you are the file's owner, you can use any file verb unless the access table explicitly assigns you a more restricted use.

For discussion of file tying and concurrent use of files, see Chapter 14, "APL Component Files". ***Watch out:*** It is poor practice to have the tie number appear as a *constant* in your programs. If you were to do that, you would run the risk that a tie number appearing in one program conflicts with a tie number of some other program that you want to have running at the same time. It is preferable to generate a tie number arbitrarily and then store it somewhere that the verbs requiring it can find it; for example, as a global noun in the workspace. You can also invoke a verb that in effect refers to files by name, matching the name you propose with the result of `⎕names`, and then selecting an appropriate element from `⎕nums`.

You could generate an arbitrary non-conflicting tie number by an expression such as:

```
tieno←((⍳1+⍴⎕nums)∈0,⎕nums)⍳0
```

The file system imposes a limit on the total number of files that one task can have tied simultaneously. When you attempt to tie more files than that limit, the system rejects the statement with the message `file tie quota used up`. There is also a limit imposed by SHARP APL on the total number of files tied (summed over all users). When you attempt to tie a file that would exceed that limit, the system rejects the statement with the message `file system ties used up`.

File permission: Any.

Result: None.

□svc
Shared Variable Control

Monad □svc reports and dyad □svc sets a name's *shared variable control.* □svc has argument rank 1.

A cell of ω is a character list containing a shared name. ω may therefore be a table with one name per row.

For dyad □svc, a cell of α is a four-element Boolean list indicating the desired state of the corresponding name in ω. α may therefore be a four-element list, or a four-column table, one state-setting per row. The four positions in an α-cell have significance as follows:

Inhibit re-set		Inhibit re-use	
You	Partner	You	Partner

The result of □svc (either monad or dyad) consists of rank-1 cells of four Boolean elements, indicating the *effective state* of a name in ω. The effective state is:

(State you set) ∨ (State your partner set)

For further discussion, see Chapter 11, "Shared Variables".

□svn
Shared Variable Clone ID

Monad □svn lets you to set or see the identifying number used to distinguish multiple tasks for the same user number ("clone identification").

ω is the numeric clone ID you propose, a deliberately out-of-range clone ID, as a numeric item, a character list, or a boxed item or list. The latter two arguments are discussed below.

The clone ID of a task is a non-negative integer. A task sets its own clone ID. The clone ID may be any non-negative integer that is not already in use as the clone ID of another task on the same user ID.

By default, a task that has not set its own clone ID is presumed to have clone ID 0. However, once one task has asserted its use of the default clone ID, 0 is no longer valid for any of the other tasks running on the same user number.

Result: The clone ID in effect after executing □svn. Hence when □svn ω is successful, the result is ω.

When there is no valid clone ID (usually, because the task has established none of its own, but another task has made us of the default clone ID 0), the result is ¯1.

When ⍵ is invalid, the result is nevertheless the clone ID in effect. Hence a deliberately invalid argument reports the current clone ID.

A convention used in many applications is to set the clone ID to the task number of the task, using an expression such as *k*←⎕*svn* ''⍴⎕*runs*. See also the discussion of clone ID in Chapter 11, "Shared Variables".

If ⍵ is a character list representing a system ID, the result is the numeric value of that system ID. The result is ¯1 if your workspace does not currently have a clone ID and the default clone ID (zero) is being used by another task with the same processor ID.

If ⍵ is a three-element boxed list of character lists (containing the system ID, user name, and user password, respectively), the result is the numeric value of the system ID or ¯1. The following examples show use of this form of ⎕*svn*:

```
      ⎕svn 'mvs05'
305050

      ⎕svn 'sys1'⊃'gibson'⊃'sg1962'
30582
```

Your active workspace can only use one numeric system ID at a time. To change the numeric system ID, you must end the link to the Network Shared Variable Processor (NSVP) on the remote system using dyadic ⎕*svn* (see below), then reassign the system ID using monadic ⎕*svn*.

Dyad ⎕*svn* is used to disconnect the link to the NSVP on a remote system. The format of dyadic ⎕*svn* is as follows:

```
nnn ⎕svn ''
```

where *nnn* is either the numeric system ID assigned by monadic ⎕*svn* or zero (0). If it is a system ID, the specified link is disconnected. If it is zero, any existing link to the issuing user number is disconnected.

□SVO
Shared Variable Offer

Dyad □SVO offers to share names with other tasks. Monad □SVO returns the degree of coupling of the names contained in ω.

Left Argument

A left-argument cell can be either an item, a one-element list, or a two- or three-column table. If α is an item or a one-element list, it specifies the processor ID of the task with which you are offering to share. If α is a two-column table, it specifies the processor ID and clone iD of that task. If α is a three-column table, it specifies the system ID, processor ID, and clone ID of that task. The clone ID is often 0, especially in offers to APs; 0 is the default if no clone ID is specified. The default value for system ID is ¯1, which indicates that the share is being offered to a processor on the same system as the task issuing □SVO.

Right Argument

A right argument cell is a character list, containing a name, or a name and its surrogate name separated by one or more blanks. Thus an offer of several names is a table, with a row for each name, or for each name and its surrogate. (For further discussion of this point, see "Surrogate Names" in Chapter 11, "Shared Variables".)

Each of the names you offer must be both valid and available. (A surrogate name must be valid, but does not have to be available.) "Available" means either the name is already in use as the name of a noun, or it is as yet unused. Offering to share an unused name implies that it will become the name of a noun, even if it is not yet. You cannot use the name for something else (for example, a verb) while the offer remains in effect.

Monad □SVO

Monad □SVO reports the status of the names in ω. The argument is the same as for dyad □SVO. However, when the original offer was made with a surrogate name, it is unnecessary (but harmless) to repeat the surrogate here.

Result:

Both for monad and dyad □SVO, the result is the *degree of coupling* of the names in ω. The result is an integer item, encoding the degree of coupling as follows:

0 The name has not been offered (or is not well formed).

1 You have offered the name, but your offer has not been matched, or your partner has retracted from a full share.

2 You have made an offer and it has been matched by an offer from the partner you proposed. Sharing is in effect.

□svq

Shared Variable Query

Monad □svq reports incoming offers addressed to you. You are most likely to need this in a program designed to run as a "server" task, such as an S-task launched by a "parent" APL task, or an S-task that serves several users concurrently.

□svq returns different results depending on the form of its argument:

□svq '' ***Empty argument.*** The result is a two-column table of the processor IDs and clone IDs of tasks that have outstanding offers to you, one processor ID pair per row. An offer is outstanding when, from the offerer's point of view, its degree of coupling is 1.

□svq <'' ***Boxed empty argument.*** The result is a three-column table of the system IDs, processor IDs, and clone IDs of tasks that have outstanding offers to you, one per row. An offer is outstanding when, from the offerer's point of view, its degree of coupling is 1. This is used to inquire about offers from other systems.

□svq ID ***two-item argument.*** When the argument is a two-item numeric list containing the processor ID and clone ID of one processor that has an offer outstanding to you, the result is a table of the names that that processor has offered to you.

□svq ID ***three-item argument.*** When the argument is a three-item numeric list containing the system ID, processor ID, and clone ID of one processor that has an offer outstanding to you, the result is a table of the names that that processor has offered to you. This allows you to share with processors on other systems.

Ordinarily, you use □svq '' to find out who is making offers. Then, if the resulting list of processor numbers is not empty, you inquire individually about each processor number in turn.

The following examples illustrate the use of □svq. Note the use of ¯1 in the result of the first example. This shows that the offer was made by a task

on the same system as your active workspace (not on a remote system). You can use ¯1 as the system ID in a three-element list argument to □*svo*; this treats names shared with tasks on your system in the same manner as names shared with tasks on a remote system.

```
      □svq <''
304050 314158 99
¯1 124 4017

      □svq 304050 314158 99
var01
apple2
revnum9
```

□*svr*
Shared Variable Retract

Monad □*svr* retracts the offer to share a name (or several names). ω is a list of names in the same form as the right argument of □*svo*. The result is a numeric list, in the same encoding as the result of □*svo*, but reflecting the level of coupling *before* the offer to share was retracted.

□*svs*
Shared Variable State

Monad □*svs* reports the *state* of a shared name (or a set of shared names). For a name you share with a partner, the state indicates which of you knows the current value of the shared name, and which (if either) remains unaware of a value that the other has set. □*svs* is used to identify where changes have occurred when a *state change* has been detected (see □*sc*). It may also be useful in tracing or debugging the process of communication between partners.

ω is a list of names in the same form as the right argument of □*svo*.

A result cell is a four-element Boolean list (so a result for several names is a four-column table with a row for each name). The four columns are interpreted in the same way as the corresponding columns of □*svc*:

1 ***inhibits:***	Set		Use	
by:	You	Partner	You	Partner
until:	Partner has used or set	You have used or set	Partner has set	You have set

There are four possible values for a result cell. The values and their interpretation are shown in Figure 16–9. See also Chapter 11, "Shared Variables".

Who has set a value that the other has not used?		Who knows the current value?		Interpretation
You	Partner	You	Partner	
0	0	1	1	You and your partner are both aware of the current value, and neither has since set the name. (This is the initial value when sharing is first established).
1	0	1	0	You have set the shared name, but your partner has not yet used it.
0	1	0	1	Your partner has set the shared name, but you have not yet used it.
0	0	0	0	The argument is not the name of a shared name.

Figure 16–9: Possible values of a `⎕svs` *result cell.*

⎕tie

Exclusive Tie

Dyads `⎕tie` and `⎕stie` establish linkage between the name of an APL component file and the tie number used to refer to it from the active workspace. `⎕tie` gives you exclusive access, whereas `⎕stie` permits other users to share the file at the same time.

'[*libnum*] *filename*' ⎕tie *tn* [, *pn*]

The right argument is numeric, containing the proposed tie number, followed (when required) by the passnumber called for in the file's access table.

The proposed tie number must be a *positive integer less than* 2^{31} and not in use by this task as a tie number of another file. Note that it does not matter what the tie number is, provided only that you choose a number that is not in use to tie some *other* file, and (as a practical matter) the applications referring to the tie number have a way to know what it is.

When the right argument has two elements, the second is the passnumber. When the argument is a scalar or one-element vector, a passnumber of 0 is understood.

The left argument is a character list (or item) containing the name of an *existing* file. If you omit indication of library, your own library is assumed.

The file named in ⍺ must be one that is not now tied by any task (including your own). An attempt to ⎕tie an already-tied file is rejected with *file tied*.

⍵ is the tie number you propose. It may be any positive integer not now in use as a tie number. If the file's access table requires a passnumber in order to ⎕tie it, you must include the passnumber as a second item in ⍵.

Effect: The file is exclusively tied. You may proceed to use the file. No other tasks may tie or share-tie the file as long as you have it tied. The tie number (and passnumber, if you included one in ⍵) must appear in the argument of any of the file access verbs.

Watch out: It is poor practice to have the tie number appear as a *constant* in your programs. If you were to do that, you would run the risk that a tie number appearing in one program conflicts with a tie number of some other program that you want to have running at the same time. It is preferable to generate a tie number arbitrarily and then store it somewhere that the verbs requiring it can find it; for example, as a global noun in the workspace. You can also invoke a verb that in effect refers to files by name, matching the name you propose with the result of ⎕*names*, and then selecting an appropriate element from ⎕*nums*.

You could generate an arbitrary non-conflicting tie number by an expression such as:

```
tieno←((⍳1+⍴⎕nums)∈0,⎕nums)⍳0
```

The file system imposes a limit on the total number of files that one task can have tied simultaneously. When you attempt to tie more files than

that limit, the system rejects the statement with the message `file tie quota used up`. There is also a limit imposed by SHARP APL on the total number of files tied (summed over all users). When you attempt to tie a file that would exceed that limit, the system rejects the statement with the message `file system ties used up`.

File Permission: `2`. However, having permission to tie a file does not of itself give you permission to do anything else. The use you can make of the tied file depends on the file's ownership and access table. When the file belongs to a user other than yourself, the file's access table specifies what file verbs you can use (see the discussion of file access in Chapter 14, "APL Component Files"). When you are the file's owner, you can use any file verb unless the access table explicitly assigns you a more restricted use.

See Chapter 14, "APL Component Files".

Result: None.

□*trap*

Event Trap Control

By setting the value of the system noun □*trap*, you specify the action to be taken if various events should occur. Events include errors, interrupts, and return to immediate execution. See also Chapter 10, "Event Handling".

□*trap* is shared with the interpreter. In general, you set its value and the interpreter uses it. (However, the interpreter sets to `''` any value that is syntactically invalid.)

□*trap* may be either a table or a list. When it is a table, it has one row for each trap definition. When it is a list, the first character serves as a delimiter, and subsequent uses of the delimiter serve to separate the various definitions. The delimiter may not be a numeral, a blank, or the character −, which has a special use in the events field.

In general, each trap definition (that is, each row of a table □*trap*, or each of the definitions separated by the delimiter in a list □*trap*) has the following form:

List of event numbers	Qualifier	Action	Recovery expression

The successive fields within a trap definition need no delimiter between them, but may be separated by any number of blanks. The fields are filled out as follows.

□trap Event Numbers Field

Events are identified by numerals. Successive numbers are separated by blanks, for example `2 14 5`. The order in which events are listed does not matter. A hyphen means "all events" in a range of numbers. For example `1-11` means "Events 1, 11, and all events in between." Figure 16-4 shows all event numbers currently defined by SHARP APL.

`0` means any event less than 1000; it has the same effect as `1-999`. Setting a trap for `1000` applies to any event greater than 1000 and less than 2000; it has the same effect as `1001-1999`. Where *no* numbers are stated, the trap definition applies to *all* events other than event 2001.[17]

□trap Qualifier Field

Either this field is empty, or it contains the letter `o`. The presence of an `o` makes the trap operate only at the verb's *own* level. That is, the trap works in the verb to which □trap is local, but not in verbs invoked within it.

Note that `o` may be needed because, when verbs invoked within the current verb also localize □trap, they may nevertheless fail to trap an event. Unless the `o` qualifier is included, this □trap may respond to an event that occurs in one of the sub-verbs.

Watch out: You must separate the qualifier from the surrounding trap definition by *at least one space.*

□trap Action Field

The action is indicated by a single letter, as follows:

`s`	***Stop***	Preempts other traps, if any.
`n`	***Next***	Exempts the current trap, leaves more global ones in effect.
`e`	***Execute***	Executes the recovery expression in the current environment.
`c`	***Cut and execute***	Cuts back the state indicator, then executes.
`i`	***Immediate***	Resumes in mid-line (after the recovery expression has supplied the missing name).
`d`	***Do***	One of four drastic measures. (See below.)

[17] Whether it applies to events 67 and 68 is moot; when they really occur, there is not much you can do about them. The trap does apply, however, if you □signal those events. □signal of these events is not recommended, however – it tends to confuse support staff and send them looking for system problems rather than application problems.

`⎕trap` Recovery Expression

The recovery expression is a line of APL in the same form as you might type it from the keyboard or assign it to `⎕lx`.

In particular, with action `e` and `c`, if you want execution of the suspended verb to be resumed, you must include the branch arrow →. However, with action `i` (where the recovery expression supplies the definition of an undefined name) execution resumes at once when the name is defined, and the recovery expression should *not* include a branch arrow.

If the recovery expression itself contains an error, that error cannot be trapped, and execution halts.

The trap actions and their associated recovery expressions are listed in the following sections.

Action `s` : Stop

Stops execution when an event occurs. This action takes no recovery expression. In effect, it restores the default action of the interpreter, and is often used during debugging.

Note that setting `⎕trap←'n s'` is stronger than `⎕trap←''` because making the visible `⎕trap` empty does not remove the action of other traps that may be shadowed.

Action `n` : Next

The action `n` instructs the interpreter to look no further in the present `⎕trap`, but to move on to the next (more global) `⎕trap`. It has the effect of nullifying trap definitions in its own `⎕trap` that appear later in the trap definition.

Like `s`, action `n` takes no recovery expression.

Action `i` : Immediate

Action `i` is usable only with event 6, `value error`. It is designed to supply the value of the undefined name and resume execution. A common use of this action is to implement demand paging systems.

Action `i` is the only action that permits resumption of an interrupted line at a point other than the beginning.

Action `i` requires a recovery expression. The recovery expression must *not* include a branch statement. It should do something to remedy the fact that an undefined name occurs in the sentence being executed (for example, read its definition from a file).

When the recovery expression has been executed, the interpreter tries to resume execution of the sentence it was executing when it found an undefined name.

If the name is *still* undefined after execution of the recovery expression, event 6, `value error` is signaled again, but the `i` action will not act upon it.

Action `e` : Execute

The interpreter executes the recovery expression in the current environment. To resume work after completing the remedial action, the recovery expression should include a sentence such as `→⎕lc`.

Action `c` : Cut Back and Execute

The interpreter aborts the execution of any verbs invoked by the verb that localized `⎕trap` (thereby cutting back the state indicator so the verb to which `⎕trap` is local is at the top), and then executes the recovery expression.

Action `c` differs from action `e` only when the operative trap is more global than the verb being executed. Especially where the recovery expression uses a branch to resume execution, it is usually necessary to use action `c` rather than `e`. That is because the recovery expression is usually written as a branch to a destination that makes sense only in the verb that set the `⎕trap`. For example, if verb `v1` has a trap whose recovery expression is

```
fixup ◇ →retry
```

the destination `retry` is perhaps a label on line 6 of `v1`. If verb `v1` invokes `v2` and `v2` encounters an event which this trap traps, the effect with action `e` might be to go to line 6 of `v2`, when what you really want is to resume at line 6 of `v1`.

Action `d` : Do

This action invokes one of several actions that are not directly attainable by executing an APL sentence (and hence not obtainable from actions `e` or `c`). In addition, action `d` is the only action permitted in response to event 2001 (return to immediate execution), which is not otherwise a trappable event.

The recovery expression may be empty, or may contain one of the following keywords:

```
exit        clear        off
```

These actions have the following effects:

`d` Action `d` with no keyword has no effect, other than to handle the event and thus prevent more global `⎕traps` from being searched.

`d exit` Causes execution to exit from the verb that contains the `⎕trap`, and return to the environment in which that verb was invoked.

`d clear` Clears the workspace. The former contents are lost.

`d off` Clears the workspace and terminates the APL session. The action `d off` terminates the APL task without saving a `continue` workspace, whereas `⎕bounce` saves `continue`, or another termination workspace name, if one was specified by `⎕twsid`.

⎕ts
Timestamp

The system noun `⎕ts` contains the current date and time of day (represented by the computer's internal clock) as a seven-item *numeric list* containing the following information:

`⎕ts[1]`	Year
`⎕ts[2]`	Month
`⎕ts[3]`	Day
`⎕ts[4]`	Hour
`⎕ts[5]`	Minute
`⎕ts[6]`	Second
`⎕ts[7]`	Millisecond

The first three elements of `⎕ts` always indicate a date, and the last four elements always indicate a time of less than 24 hours.

⎕twsid
Termination Workspace ID

This system verb establishes and reports the name of the workspace to be saved in the event of an abnormal termination of the APL task.

The active workspace of a task that was terminated by an external event is saved by SHARP APL whenever possible. For T-tasks and S-tasks, the name of the saved workspace is `continue`. For B-tasks and N-tasks, the

name is determined from the parameters set with ⎕*run*. ⎕*twsid* provides control over this behavior, as follows, by allowing the task to:

- Inquire whether the active workspace is to be saved if the task terminates abnormally.
- Inquire what name the active workspace will be saved under.
- Specify that the active workspace is not to saved if the task terminates abnormally. Also, to revoke this state.
- Specify the workspace ID to be used for saving the active workspace if the task terminates abnormally. This workspace ID can include a password on the workspace or a library in which to save the workspace. The library must belong to the user number that the task was run on.

Monad ⎕*twsid* has the following format and definition:

```
⎕twsid ω
```

where ω is one of the following:

`1` Save the active workspace using the current termination workspace ID.

`¯1` Do not save the workspace.

`0` Return the current setting of ⎕*twsid* (`1` or `¯1`).

`''` Return the current termination workspace ID.

charlist Set the termination workspace ID to that specified in *charlist*. The workspace ID may be one of the following:

- A valid workspace name, such as *deadandgone*.
- A valid workspace name, followed by a colon and a password, such as *deadandgone*`:`*buried*.

The termination workspace ID can also be defined in terms of information that is associated with the task. For example, the statement:

```
⎕twsid 'byebye',⍕1⌷⎕runs
```

sets the termination workspace ID to *byebyeXXXX*, where *XXXX* is the task ID.

To save a termination workspace with a name other than *continue*, the user's workspace quota must allow the user to save at least one more workspace.

The result of ⎕*twsid* with a valid numeric argument is the previous setting of the numeric value of ⎕*twsid*. The result of ⎕*twsid* with either an empty list or a valid workspace name as its argument is a 22-character list containing the library number and workspace name of the previous setting of the termination workspace ID. The password, if any, is not returned.

□*ul*

User Load

This system noun reports the number of APL tasks (of any type) now signed on to the SHARP APL system.

□*untie*

Untie Tied Files

Monad □*untie* unties the files whose tie numbers are included in its argument. ⍵ is a numeric list (or item) of numbers currently in use to tie files. Passnumbers are *not* included in ⍵.

The right argument is a numeric list (or item) of tie numbers.

Passnumbers do not apply to □*untie*. You do not include a passnumber even if you did specify one to tie the file. Any number appearing in the right argument ⍵ is assumed to represent a tie number.

Effect: The files whose tie numbers appear in ⍵ are untied. Once you have untied a file, when that tie number appears in the argument of a file verb, the file system rejects the sentence with the message *file tie error*. There is no way to refer to the files that were untied until you subsequently re-tie them.

File permission: None.

Result: None.

□*vi*

Verification of Input Format

Monad □*vi* reports whether each field in ⍵ is a valid numeric field.

⍵ is a character list (or item). □*vi* treats ⍵ as a sequence of fields, in the same fashion as □*fi* (described earlier in the chapter).

The result is a Boolean list containing a 1 for each field that represents a well-formed number, and 0 for each field that does not.

□*vi* can be used in conjunction with □*fi* to provide a validity check on the input character list, as described in the discussion of □*fi*.

□wa
Work Area Available

The system noun □wa is a numeric item indicating the current amount of work area available in the active workspace (in bytes). It is computed by subtracting from the gross size allocated to the workspace the amount currently in use for the definitions of nouns and verbs, the symbol table, and the execution stack and partial results associated with it.

□ws
Workspace Information

Dyad □ws reports a number of different items of information about the current state of the workspace. The information returned in the result depends on the values of the two arguments, each of which is a numeric item. The possible values and the results associated with them are as follows:

1 □ws ⍵ ***Namelists.*** ⍵ is a *numeric item* identifying the class or classes of names to be included in the result.[18] ⍵ must be a *single item* whose value is the sum of the numbers representing the desired classes. The classes are identified by powers of 2:

1 Verb
2 Noun
4 Group
8 Label

The result is a character table containing one row for each visible name in any of the classes identified in ⍵.

2 □ws 1 ***Workspace ID.*** The result is a character list containing the name of the active workspace. When the workspace has not been named, the list is empty. Otherwise, it has 22 characters, in the same format at returned by □lib for the names of files. Although the format differs slightly, the information conveyed is the same as that displayed by the command)wsid.

2 □ws 2 ***State indicator.*** The result is a character table containing one row for each item on the state indicator, in the same format as displayed by the command)si. Each row contains the name of a user-defined verb, followed by a line number in brackets. A verb that is suspended is marked by as an asterisk following the closing bracket of the line number. The line number is the number

[18] The effect of 1 □ws ⍵ is essentially the same as that of □nl. In both cases, ⍵ identifies the classes to be included in the result, but 1 □ws identifies them by a scheme different from that of □nl.

of the line currently active (for a verb that is not suspended) or the number of the line next to be executed (for a verb that is suspended). The rows of the table show the most recently invoked verbs at the top. Calls to ⍎ or to ⍞ input are shown in the result in the same way as user-defined verbs; each has a row on which the symbol ⍎ or ⍞ appears alone.

`2 ⎕ws 3` ***Workspace environment.*** The result is a 12-element numeric list. A number of its elements are no longer in use and contain `¯1` or `0`, but are preserved to maintain consistent position for the others. The meaning of an element at each position in the result is as follows:

`(2 ⎕ws 3)[1]` Unused; value `¯1`.

`(2 ⎕ws 3)[2]` APL shutdown pending. This value is normally `0`. When its value changes to `1`, SHARP APL is in the process of shutting down. See `⎕sc`. When shutdown has been initiated, new tasks may not be started, and service facilities (such as HCPRINT and the Batch Task Scheduler) may be withdrawn from service. User tasks will be interrupted soon thereafter.

`(2 ⎕ws 3)[3]` Effective workspace size. This reports the actual workspace size in bytes that is under the control of the APL application. This is useful information in the implementation of demand-paging schemes for APL objects in the workspace.

`(2 ⎕ws 3)[4]` Unused; value `¯1`.

`(2 ⎕ws 3)[5]` `1` in a task using *new* character control; `0` otherwise.

`(2 ⎕ws 3)[6]` `1` in a *recovered* workspace; `0` otherwise.

`(2 ⎕ws 3)[7]` Current size of the symbol table.

`(2 ⎕ws 3)[8]` Symbols in use.

`(2 ⎕ws 3)[9]` User number of owner (same as `1↑⎕ai`).

`(2 ⎕ws 3)[10]` Node and line number of task.

`(2 ⎕ws 3)[11]` Task ID.

`(2 ⎕ws 3)[12]` Encoding used by session handler for type of task:

`0` N- or B-task
`1` Obsolete; was T-task, IBM

(Correspondence)
2 Obsolete; was T-task, IBM (BCD)
3 T-task, old-style ASCII without APL
4 T-task, ASCII, with APL
5 T-task, ASCII, with APL (bit-paired)
6 T-task, Telex
7 S-task
8 T-task, IBM/PC
9 T-task, new-style ASCII without APL

2 □*ws* 4 ***Workspace information.*** The result describes the workspace by a three-element numeric list (in the same way that □*rdci* describes a component of an APL file):

(2 □*ws* 4)[1] Bytes needed to store the contents of the workspace.

(2 □*ws* 4)[2] User number of the task that saved the workspace.

(2 □*ws* 4)[3] Timestamp as 60ths of a second since 1960-03-01 00:00:00.

3 □*ws* ⍵ ***Group members.*** ⍵ is the name of a group. The result is a character table containing the names that are members of the group.

4 □*ws* ⍵ ***Storage space.*** The number of bytes of workspace storage required to store each of the objects named in ⍵. ⍵ is a namelist, either a list with successive names separated by blanks, or a table with one name per row. The result is the number of bytes that would be required to store the named object if it were the only instance in the workspace. (In fact, an object may be stored merely by pointing to another instance of the same value). The value includes space occupied by the header that the interpreter uses to describe the object's type, rank, and shape, but not the space occupied by the entry for its name in the symbol table. If a name is that of a shared name, the result does not constitute a reference to the name.

5 □*ws* ⍵ ***Name class by stack level.*** ⍵ is a namelist, formed either as a list with successive names separated by blanks, or as table with one name per row. The result is a *numeric table*, having one row for each name in ⍵, and one column for each level of the state indicator, plus an additional final column representing the global environment (not represented in the state indicator). Values within the result as follows:

- `¯1` Not localized at this level
- `0` Localized at this level, not used
- `1` Localized at this level, used as a verb
- `2` Localized at this level, used as a noun
- `4` Group
- `8` Localized at this level, used as a label

`6 ⎕ws ω` ***Noun Value.*** *ω* is the name of a noun. The result is the value of the noun, or `domain error` if the name is not that of a noun. If the name is a shared name, it does not constitute a reference to the name.

⌶ I-Beam Verbs

In APL\360, a miscellany of system-reporting functions was provided by the catch-all primitive `⌶`, named *I-beam.* This family was subsequently withdrawn from IBM APL systems. SHARP APL continues to tolerate `⌶`, but it is considered to be obsolescent; descriptions are included here mainly to help in the maintenance of existing programs that still use `⌶`.

`⌶19` *Keying time* in 60ths of a second. Cumulative time since start of task during which it has been available for input. Approximately `⎕ai[4]×60÷1000`.

`⌶20` *Time of day*, in 60ths of a second since 00:00:00 of the day on which the task started.

`⌶21` *CPU units*, cumulative since start of task, in 60ths of a unit. Approximately `⎕ai[2]×60÷1000`.

`⌶22` *Work area.* Remaining free space in the active workspace, in bytes. Equivalent to `⎕wa`.

`⌶23` *User load.* Number of current tasks. Equivalent to `⎕ul`.

`⌶24` *Sign-on time*, in 60ths of a second since 00:00:00 of the day when the task started.

`⌶25` *Sign-on date.* Calendar date at start of the task, encoded as `100⊥M,D,100|Y`.

`⌶26` *Current line.* Equivalent to `''⍴⎕lc`.

`⌶27` *Line counter.* Equivalent to `⎕lc`.

`⌶28` *Session management,* encoding as:

	¯1 N- or B-task
	0 S-task
	1 Obsolete; was IBM 2741 Correspondence
	2 Obsolete; was IBM 2741 BCD
	5 ASCII
`⌶29`	*User number.* Equivalent to `''⍴⎕ai`.
`6⌶11 0`	*Walkaway.* Condition the current active workspace so that it will sign off the task with `)continue hold` at next return to immediate execution. Result: `0`. At present no equivalent `⎕trap` setting.
`6⌶11 1`	*Comeback.* Nullify earlier `6⌶11 0`. Result: `0`.
`6⌶11 2`	*Clearout.* Condition the current active workspace so that it will execute `)clear` at next return to immediate execution. Result: `0`. Equivalent to `⎕trap←'⍝ 2001 d clear' .`
`6⌶11 3`	*Noclear.* Nullify earlier `6⌶ 11 2`. Result: `0`.
`6⌶12 1`	Same effect as `)keyb`.
`6⌶12 3`	Same effect as `)keyb nomsg`.

17 *System Commands*

Chapter 2, "Tasks, Users, Libraries" introduced the system commands and their place in an APL system. This chapter provides an individual description for each of them.

Every system command begins with a right parenthesis), followed by the name of the command and its arguments (if any). The names of commands may be spelled in either the first or second alphabet (that is, either lowercase or capitals).[1] However, in the names of workspaces, passwords, or the names of nouns or verbs in an APL workspace, case is significant; where such names appear in the arguments of a command, you have to use the alphabet appropriate to the name in question.

The various commands differ in syntax. Where a command takes more than one argument, the arguments are separated by blanks. Names that appear in the arguments to a system command are never surrounded by quotes (even when related verbs may require quotes around their arguments).

You may enter a system command only when the interpreter is in immediate execution mode (including □-input mode), in response to the interpreter's standard prompt of *newline* followed by six blanks. You may not execute a system command from within the ∇-editor. A system command may not be part of a user-defined verb. (However, a program that controls another APL task by way of a shared name may use the shared variable interface to transmit to the other task the characters that form a system command, provided the other task is in immediate-execution mode when it receives the command.)

Figure 17–1 is a list of valid system commands, with optional portions of a command shown surrounded by square brackets. Note that the square bracket character itself is never part of any command.

[1] This manual shows the first alphabet as lowercase. Some older terminals, or a session with `)cc reverse`, may show capitals for the first alphabet.

) [*n*]	Recall previous line [edit at *n*]
)*nnnnnnn*:*pass* [:*newpass*]	Sign on with password [change password]
)`blot` [*n*]	Conceal input
)`cc` [*normal*|*reverse*]	Character control (alphabet selection)
)`clear`	Clear workspace
)`continue` [`hold`] [:*newpass*]	Save ws for resumption, and sign off
)`copy` [*libno*] *ws* [:*pass*] [nm_1 nm_2 ...]	Copy from saved to active ws
)`cpulim` [*trans* [*cum*]]	Limit usage
)`drop` [*libno*] *ws* [:*pass*]	Discard saved workspace
)`erase` [$name_1$ [$name_2$...]]	Erase global objects
)`fns` [*firstname*]	List verbs [starting with *firstname*]
)`group` *grpname* [$name_1$ $name_2$...]	Form or disperse a group
)`grp` *grpname*	List members of a group
)`grps` [*firstname*]	List groups [starting with *firstname*]
)`keyboard` [`lock` | `nomsg`]	Lock keyboard or bar messages
)`lib` [*libno*] [*firstname*]	List workspaces in library
)`load` [*libno*] *ws* [:*pass*]	Load and activate saved workspace
)`msg`[n] *tasknum text*	Send message to another task
)`nlt` [*lang*]	Inquire or set national language
)`off` [`hold`] [:*newpass*]	Sign off [hold line] [change password]
)`opr`[n] *text*	Send message to SHARP APL operator
)`pcopy` [*libno*] *ws* [:*pass*] [$name_1$ $name_2$ ...]	Protected copy from saved to active ws
)`port` [*usr* | *nnnnnnn*]	Report port and task ID
)`reset`	Synonym for)`sic`
)`save` [*libno*] [*ws*] [:*pass*]	Save a copy of the active workspace
)`seal`	Seal the active workspace
)`si`	State indicator
)`sic`	State indicator clear
)`sinl`	State indicator with namelist
)`siv`	Synonym for)`sinl`
)`symbols` [*n*]	Size of symbol table
)`terminal` [*type*]	Set terminal type
)`vars` [*firstname*]	List nouns [starting at *firstname*]
)`wsid` [*newname*]	Save-name of the active workspace
)`xload` [*libno*] *ws* [:*pass*]	Load workspace without executing ⎕`lx`

Figure 17–1 : List of System Commands.

)

Recall Previous APL Line

)[*n*]

When you enter) alone on a line, the interpreter displays the line of APL most recently entered from the keyboard during immediate execution. You may accept the line as is, or edit it as if you had just entered it yourself. When you press *Enter*, the line goes to the interpreter for execution.

This is the only system command that you may use while in definition mode with the ∇-editor. In that context, the line of APL most recently entered from the keyboard is copied as a line of the verb now being defined. You may accept the line as is, or edit it as if you had just entered it yourself. When you press *Enter*, the line becomes a part of the verb you are defining.

Optionally, you can follow) by a number between 0 and □*pw*. The number after the) has the same effect as the number after the □ in the ∇-editor. That is, it invokes the line editor. 0 invokes the one-pass line editor, so its effect is no different than) alone. A number greater than 0 but less than □*pw* invokes the two-pass editor. A number greater than □*pw* is illegal, and produces the message `defn error`. (See the description of the ∇-editor in Chapter 8, "Verb Formation".)

The line thus recalled is the one you most recently entered *at the current level of execution*. When the interpreter starts work on a line entered from the keyboard, it creates an added level of execution. That level remains in effect until execution of that verb comes to a normal end or is abandoned. Thus, when you execute the) command while you have nothing on the execution stack, you get the line you most recently entered at that level.

While execution of a user-defined verb is suspended, the) command shows you the line most recently entered *during suspension of that verb;* a line entered earlier is shadowed until you complete or abandon the current verb's execution and return to the earlier level.

In recalling a previously entered line, SHARP APL refers to the execution stack in the active workspace. The stack is included when you save an active workspace. If a workspace was saved during suspended execution, when you subsequently load it, after clearing its current level of execution, the) command will recall keyboard entries that preceded the save.

The line recalled is always an APL line that has been passed to the APL interpreter. You *cannot* use this mechanism to recall data entry in response to a request for ⍞-input, or an earlier system command, an empty line, or one consisting solely of a comment.

) *nnnnnnn:pass*

Sign-on to SHARP APL

)*nnnnnnn:pass* [*:newpass*]

At a time when your terminal is connected to the SHARP APL system but not signed on, this command signs on a new T-task. For ***nnnnnnn***, substitute your user number. The colon is required. For ***pass***, substitute your password.

If you wish, you may enter a second colon and a new password. It will become the password required at your next sign-on.

In response, SHARP APL typically displays a message showing the task number of the new interactive task thus started, the date and time, and the name associated with your user number. It may also display a one-line message (known as the "hi" message) sent from the SHARP APL system operator to all interactive users as they sign on. Finally, it displays the system *banner*, a message identifying the particular SHARP APL system or installation. (See also the discussion and examples under "Starting a T-Task", in Chapter 2, "Tasks, Users, Libraries".) Display of the above information is site-dependent. In the case of *constrained users*, it is typical to suppress the entire display.

If your sign-on transmission includes characters not permitted in a sign-on, or includes characters but not a colon, the SHARP APL system rejects it with the message `incorrect signon:`. If your sign-on is well formed, but includes an invalid user number or an incorrect password, the SHARP APL system rejects it with a message such as `number not in XXX system` (replacing `XXX` by the name of the installation). This message is site-dependent, and hence may vary depending on the system you are using.

)*blot*

Security Blot

`)blot` **[*n*]**

This command produces a blot to mask confidential information, such as a sign-on or a password. The blot is a single line, and can be no longer than □*pw*. You can include a number that specifies the length exactly. When you omit the number, the blot is 35 positions long.

At printing terminals, the blot consists of a pattern of overstruck characters to mask what you enter. At some video terminals (for example, the IBM 327x), the blot is a "no-print" field.

SHARP APL accepts a blot command from a terminal that is not yet signed on. SHARP APL usually sends an unsolicited blot command to a terminal that seems active but is not signed on, but in case you did not get one, you can ask for one explicitly by entering a line consisting of a parenthesis.

)cc
Character Control

)cc [*normal*|*reverse*]

This command permits you to adjust the way a non-APL terminal assigns the APL alphabet to lowercase and capitals. It has no effect on an APL terminal. There are two possible arguments: *normal* or *reverse*. With no argument, the command reports the current setting.

At an ASCII terminal without APL support, electing)cc *normal* has the following effect:

- (2□*ws* 3)[□*io*+4] is 1.
- (2□*ws* 3][□*io*+11] changes from 3 to 9.
- The primary (first) alphabet is lowercase in both input and output.
- The secondary (second) alphabet is capitals in both input and output.
- In displays sent to the terminal, the third alphabet is displayed as lowercase (so that it is indistinguishable from the primary alphabet). There is no way to enter third alphabet from the terminal.
- Backspace is destructive. That is, backspacing over a character deletes it. There is no way to form composite characters by overstriking.
- Characters that the device presumably cannot display, such as APL symbols, are displayed as ? in the output.
- As shown in Figure 17–2, input characters usually provided on an ASCII device are mapped to the characters that resemble them in APL. For example, the tilde accent is treated as an APL tilde, the circumflex accent is treated as an APL ∧, the asterisk as an APL star, and so on. There is no way either to enter or to display characters (such as □ or ÷) that are not found on a standard ASCII keyboard or display.

The current setting of)cc is also reported by:

```
(2 □ws 3)[□io+4]
```

In the result, 1 means "normal" and 0 means "reverse."

The terminal transmission encoding state, which can be determined by

```
(2 □ws 3)[□io+11]
```

distinguishes between 3 (reverse ASCII without APL) and 9 (normal ASCII without APL).

The verbs *cc* and *Cc*, in workspace 1 *wsfns*, can be used to alter the character control setting under program control.

For discussion of the treatment of characters on various devices, see Chapter 15, "APL Characters".

Zero-Origin ⎕av Index	Character
4	$
14	[
15	]
16	(
17	]
18	;
19	/
20	\
26	+
27	-
30	*

Zero-origin ⎕av Index	Character
33	^
36	<
38	=
40	>
47	,
48	!
53	?
54	~
61	_
82	}
83	{

Zero-origin ⎕av Index	Character
50	.
52	blank
53	'
54	:
197	¦ NU-6
198	` NU-7
199	# NU-8
200	@ NU-9
201	" NU-10
206	%
207	&

Figure 17–2 : APL characters on an ASCII terminal with)cc normal.

)clear

Clear Active Workspace

)clear

Clears the active workspace and sets all system nouns except the *session nouns* ⎕sp and ⎕ht to their default values.

See also ⎕trap action *d*, which clears the workspace when certain events are encountered.

Conditions in a Clear Workspace

Clearing the workspace restores several parameters to their default values. These are summarized in Figure 17–3.

Parameter	*System Noun*	*Value*
Comparison tolerance	⎕*ct*	1.4196976927394189e¯14
Format control	⎕*fc*	'.,**_¯'
Horizontal tabs	⎕*ht*	Unaffected
Index origin	⎕*io*	1
Line counter	⎕*lc*	⍳0
Latent expression	⎕*lx*	''
Tie numbers in use	⎕*nums*	Unaffected
Names of tied files	⎕*names*	Unaffected
Printing precision	⎕*pp*	10
Position and spacing	⎕*ps*	¯1 ¯1 0 1
Page width	⎕*pw*	132
Random link	⎕*rl*	16807
Session parameter	⎕*sp*	Unaffected
Work area available	⎕*wa*	As set by system administrators
Shared names		None; previous offers are retracted
Symbol table		Space for 256 symbols (can vary with space available)
Workspace name		None; reported as *clear ws*

Figure 17–3 : Conditions in a clear workspace.

)*continue*

Terminate APL Session for Restart

)*continue* [*hold*] [:***newpass***]

This command terminates your APL task and usually saves the active workspace in your library under the name *continue* (overwriting any workspace of that name saved there earlier). When the task from which you thus signed off was a T-task, at the next occasion you sign on a T-task,

SHARP APL will automatically load your `continue` workspace. (Thus, you *continue* the previous work session.)

However, if the system verb `⎕twsid` is invoked with `¯1`, no workspace will be saved. The `)continue` command effectively turns into `)off`. Furthermore, if `⎕twsid` has been used to specify a termination workspace name and `⎕twsid` was invoked with `1`, the workspace saved will be saved under the name indicated by `⎕twsid`, and will *not* be loaded automatically at the next sign-on.

Your own `continue` workspace appears to you to be unlocked. However, any other user must supply your current sign-on password as a password for the workspace. Furthermore, termination workspaces other than the one named `continue` are not subject to this protection.

The optional parameter `hold` and a colon to set the password work in the same way for `)continue` as for the command `)off` and are described there.

)copy

Copy Objects into Workspace

`)copy` [***libno***] ***ws*** [***:pass***] [$name_1$ $name_2$ ...]

)pcopy

Protected Copy into Workspace

`)pcopy` [***libno***] ***ws*** [***:pass***] [$name_1$ $name_2$ ...]

This command copies global objects from the saved workspaces ***ws*** so that they become global objects in your active workspace. When the saved workspace belongs to a library other than your own, its name must be preceded by a library number.

When no other names follow ***ws***, all global user-defined names in the saved workspace are copied, but not system nouns. When other names follow ***ws***, only the objects named are copied, but the list may include system nouns. If your command includes names that are not in fact the names of global objects in the workspace ***ws***, SHARP APL displays the message `not found:` followed by a list of the names not found.

After a successful copy, the list of names of global objects in the active workspace is the union of those previously in the active workspace and those in the saved workspace. The two forms of *copy* (that is, `)copy` and `)pcopy`) differ in the way they treat names that occur in both workspaces. For `)copy`, if global objects have the same name in both workspaces,

the version from the saved workspace overwrites the version in the active workspace.

For `)pcopy`, global objects already in the active workspace are *protected*; that is, they remain as they were because names from the saved workspace do *not* overwrite those already present. If any objects were protected in this way, SHARP APL displays the message `not copied:` followed by a list of the names of objects that were not copied.

Note that `)copy` and `)pcopy` deal with global objects only. They cannot copy the values of names that are localized in the saved workspace, and the values that materialize in the active workspace are global as well.

If objects are `)copy`'d into a *sealed* workspace, the effect is as if the command was `)pcopy`.

)cpulim

CPU Limit

`)cpulim` [***trans*** [***cum***]]

This command lets you set a limit on the amount of CPU time an interactive task can use without being interrupted. `)cpulim` is a useful guard against runaway code. There are two optional parameters:

- ***Transaction limit: trans.*** Maximum CPU units per immediate-execution entry (from the keyboard or through the controlling shared name of an S-task).
- ***Cumulative limit: cum.*** Maximum CPU units cumulative from the time the limit was set.

When SHARP APL notes that the limit has been exceeded, it signals event 1003, `interrupt`. Execution may halt in mid-line, which may impact resumption of execution of that line. It is possible to trap this interrupt (but of course if you do, you may be defeating your purpose in setting it).

The default setting is no limit, represented by `0 0`.

`)cpulim`	Reports the current setting.
`)cpulim` n_1	Sets the limit per entry to n_1, but no cumulative limit, and reports the former limits.
`)cpulim` n_1 n_2	Sets the limit per entry to n_1, the cumulative limit to n_2, and reports the former limits.

)drop

Delete a Workspace

)drop [*libno*] *ws* [:*pass*]

Discards the saved workspace named *ws* from storage. The active workspace is unaffected, and so is its name (which may be the same as the name of the workspace thus dropped). SHARP APL reports the date and time at which you dropped the workspace. If the workspace is locked, you need its password in order to drop it. You must be the owner of a workspace in order to drop it.

When you know you want to drop a locked workspace – for example, one called LockedWS – but you cannot remember the lock, the following will serve:

```
)clear
)wsid LockedWS
)save
)drop LockedWS
```

)erase

Erase Global Objects

)erase [$name_1$] [$name_2$ ...]

This command erases (expunges) the *global* definitions of the user-defined objects named in the list to the right. When an argument to)erase is the name of a group, both the group *and* all members of the group are erased.

Erasing a name that is shared with another task retracts the offer to share that name.

SHARP APL reports only the names of those objects explicitly mentioned in the argument that it did *not* erase. It displays their names in one or two messages, with the captions not found: or not erased:, as appropriate.

An object is not erased if its name:

- is not the name of a global object
- refers to an active verb (that is, a verb whose name is in the state indicator)
- refers to a system noun
- refers to a locked user-defined verb *and* the active workspace is sealed

)fns

Display Verb Names

)fns [***firstname***]

This command displays a list of names whose visible referent is a user-defined verb (function), in alphabetical order, horizontally across the screen. For alphabetization, a collating sequence is assumed in which all first-alphabet letters precede any second-alphabet letters.

When the command is followed by a name, the list shows only those names that occur in the alphabetized list at or after the position of that name.

The names listed are the same as those returned in the result of `⎕nl 3` or `1 ⎕ws 1`.

)group

Alter a Group

)group ***grpname*** [***name***$_1$ ***name***$_2$ ...]

This command forms or disperses a group. When you form a group, the first name is the name the group is to have, and the names that follow are the names of its members.

A group provides a shorthand way of referring to a set of global names, primarily so they can be either copied or erased without having to mention them individually in the argument to the command)copy or)erase.

The name for the proposed group must be available (that is, have no global use as the name of an APL object). The names of the members must be well-formed names, but there is no requirement that objects with those names exist.

Once a group has been formed, its name is no longer available as the name of a global noun or verb.

To add new members to an existing group ***grpname***, the syntax is:

)group ***grpname grpname newname***$_1$ ***newname***$_2$ ...

Watch out: The command)group ***grpname*** with *no* names following it creates a group with no members, and thereby disperses the group. Do not confuse)group with the command)grp, which reports the membership of a group.

Groups are obsolescent. Boxes or packages provide effective means to store or move disparate objects under a single name. While, for compatibility

with old applications, groups are still provided in SHARP APL, you should avoid proliferating them or relying on them.

)grp

Display Group Member Names

)grp ***grpname***

This command returns the list of names that are members of the group ***grpname***, in the same format as the report generated by the command)fns (above).

)grps

Display Names of Groups

)grps [***firstname***]

This command returns the names of all groups in the active workspace, in the same format as the report generated by the command)fns (above).

)keyboard

Alter Keyboard State

)keyboard [lock | nomsg]

The)keyboard command establishes the state of your terminal during the interval between entries from the keyboard. By default, the keyboard is "unlocked." You may elect to lock it. You may also elect to bar incoming messages from other terminals.

For most purposes, SHARP APL interacts with a terminal in a mode called reversing half-duplex. That is, the two sides take turns. When you are entering a line from the keyboard, SHARP APL cannot display. When SHARP APL is displaying, you cannot enter. The issue is timing: when SHARP APL completes a display, should it start listening for your input immediately, or wait until you signal that you are ready to enter?

Unless you use the)keyboard command to specify otherwise, SHARP APL assumes that you will want to enter something immediately. As soon as its display has finished, it routinely signals that it is ready for further input. (Depending on the type of terminal, it may sound the "bell" to let you know it is ready and/or it may physically unlock the keyboard on terminals that lock.) This has advantages and disadvantages.

- ***Advantage:*** You do not have to request to enter; as soon as it completes each display, the terminal is always ready.
- ***Disadvantage:*** Your keyboard is open all the time, even when you are not actually entering, and you cannot receive displays sent to you from the computer. Also, you cannot receive messages sent by the operator or other users via the `)msg` command. For example, you might not be notified of a pending system shutdown.

`)keyboard lock` causes SHARP APL to leave your keyboard locked until you specifically request to unlock it by signaling *Attention.*[2] Not until you make this explicit request does SHARP APL send its prompt for input. `)keyboard lock` is useful primarily when you expect someone to send you a message, and wish to leave your terminal free to receive it.

`)keyboard nomsg` prevents you from receiving messages at all. With this command, you can assure that a message from another user or from the SHARP APL operator will not appear in your display. (This may be important when you are producing a final report from an interactive terminal.) Someone who attempts to send you a message with the command `)msg` or `)msgn` sees the response `incommunicado`.

`)keyboard` used alone, with no argument, ends either of the states started by the other two forms of the command, and returns the terminal to normal operation.

The verb `keyb` in workspace `1 wsfns` may be used to alter the keyboard state under program control.

)lib
Display Saved Workspace Names

`)lib` [***libno***] [***firstname***]

This command causes SHARP APL to display a list of the names of saved workspaces, in alphabetical order (alphabetized as described for `)fns`, above).

You may include either or both of two optional arguments. The first identifies the library about which you are asking by its *library number.* Use it to ask about a *public library* (one whose user number is less than 1000). With no library number, the command shows names in your own library of saved workspaces. SHARP APL rejects a request for the private library of another user with the message `improper library reference`.

[2] This applies equally to an S-task. There is a *break* signal you set in the controlling shared name.

The list of workspace names is displayed in alphabetical order (following the same sequence described for `)fns`, above. When you include the optional ***firstname***, the display includes only names alphabetically at or after ***firstname***.

)load

Activate a Workspace

`)load` [***libno***] ***ws*** [***:pass***]

This command replaces the current active workspace by a duplicate of a saved workspace. The former contents of the active workspace are lost.

The new contents (and also the visible session nouns `□sp` and `□ht` brought forward from the preceding active workspace) must fit within the established size of your active workspace. This usually presents a problem only if you import a workspace from a system that used a larger workspace size.

Following the `)load` command, the interpreter immediately executes the latent expression stored as the visible value of the system noun `□lx`.

The verbs `□load` and `□qload` produce the same effect as the command `)load`; they are described in Chapter 16, "System Nouns and Verbs".

The)xload Variant

The variant command `)xload` is similar to `)load`, except that the interpreter does *not* execute the latent expression, provided the workspace is one that you own: that is, a workspace from your own private library, or a workspace that you saved in a public library. This is intended as a convenience in doing maintenance to a workspace that would otherwise start executing as soon as loaded.

)msg

Send Message with Reply

)msg ***tasknum text***

)msgn

Send Message with No Reply

)msgn ***tasknum text***

)opr

Send Message to Operator

)opr ***text***

)oprn

Send Message to Operator with No Reply

)oprn ***text***

With the command)msg or)msgn you can send a one-line message to the terminal of another person who is currently signed on. The commands)opr or)oprn send a message to the SHARP APL system operator.

You address a message to a *task number*. The task number comes after)msg or)msgn and before the message text, separated from them by blanks. (The command)port helps find a user's current task number.)

Following the task number, you may write any text, provided only that it is composed entirely of characters that SHARP APL recognizes. (The command does not require quotes around the message text.)

The commands)opr and)oprn work in exactly the same way as)msg and)msgn, except that the recipient is the SHARP APL system operator, and you do not include a task number.

If you find you are unable to sign on but you are able to contact SHARP APL (as would be evident if you are able to receive a message such as *number not in system* or *number locked out*), you may still send *one* message to the operator with the command)opr. You cannot make any further entry until the operator replies.

Message Delivery

While the person to whom you are sending the message has the keyboard unlocked (see the discussion of the)keyboard command, earlier in this

chapter) to receive input, your message cannot be delivered. It is delayed until after the next time the recipient presses *Return*. During that interval, your keyboard remains locked. If you get tired of waiting, you can press *Break*, *PA1*, or *Attention*. SHARP APL discards your message and sends you the report `message lost`.

When it dispatches your message to the recipient's terminal, SHARP APL sends you the confirmation `sent`.

If you address a message to a task that is not now signed on, SHARP APL reports `signed off` and discards your message. If the task to which you address the message is not an interactive task, or is an interactive task at which the user has elected `)keyboard nomsg`, SHARP APL sends you the report `incommunicado` and discards your message.

At the recipient's terminal, SHARP APL displays your task number, followed by a colon and then the text you entered. The character `R` is inserted between the colon and your message when the sender expects a reply (see below).

A message may contain up to 124 characters; after the blank that separates the addressee's task number from the text, additional blanks are part of the text. A message longer than that is rejected as an `incorrect command`. Messages are subject neither to the sender's nor to the recipient's `⎕pw` (so your message may not fit within the display area of the recipient's terminal).

If the recipient signals an interrupt (for example, by hitting the *Break* key) while SHARP APL is displaying your message, the display at the recipient's terminal is cancelled. However, you will not know that; the confirmation `sent` means that the message was dispatched, but not necessarily that it was received in full.

Message to Which a Reply Is Expected

If you expect the recipient of your message to send you a message in reply, it is courteous to leave your terminal in a condition ready to receive the reply promptly. The reply cannot be delivered while your keyboard is open for input (and your correspondent cannot do anything else while waiting for you to read the reply). For this reason, following a message sent with the command `)msg`, SHARP APL leaves your keyboard locked until you receive a message. If you get tired of waiting for a reply, you can unlock the keyboard yourself by signaling an interrupt.

When you do not expect a reply, send your message with the command `)msgn` rather than `)msg`. The `n` stands for "no reply." When you do that, the recipient does not see the character `R` at the start of your message, and your terminal is unlocked as soon as your message is sent.

You send:	`)msg 9482 What file has your data? /James`
Recipient sees:	`1245R: What file has your data? /James`
You send:	`)msgn 9482 Analysis is ready.`
Recipient sees:	`1245: Analysis is ready.`

)nlt
Set Language

`)nlt` **[*lang*]**

The command `)nlt` concerns the *national language table*. Used with no argument, the command reports the current setting. Used with the name of one of the supported languages, SHARP APL sets the requested language and reports the former setting.

The national language table contains messages the SHARP APL system generates in response to system commands and messages to describe events (errors or interrupts). The messages are listed in Chapter 18, "Messages".

The argument to `)nlt` is currently restricted to one of the following:

```
english
deutsch
francais
```

The effect is to change the language of the message text displayed at the terminal and included in `⎕er`. For example:

```
      )nlt english
was deutsch
      2÷0
domain error
      2÷0
       ^
```

```
      )nlt francais
c'etait english
      2÷0
erreur de domaine
      2÷0
       ^
```

```
      )nlt deutsch
war francais
      2÷0
unzulaessiges argument
      2÷0
       ^
```

)off

Terminate APL Session

)off [hold] [***:newpass***]

This command terminates an APL session. SHARP APL discards the active workspace, retracts any offers to share names, and unties any tied files.

When you include the optional parameter *hold*, the session handler keeps you connected to SHARP APL, so that you can initiate another session. If you elide *hold*, the session handler returns control to the network solicitor.

When you include a colon, the characters following the colon become your sign-on password effective the next time you sign on. (When you are doing this on a printing terminal, it is appropriate to use the)blot command before you sign off.)

A password may be up to eight characters in length, and may contain any letters or numbers.

)opr

Send Message to Operator

)opr ***text***

)oprn

Send Message to Operator with No Reply

)oprn ***text***

These commands are variants of)msg, and are discussed under that heading, earlier in this chapter.

)pcopy

Protected Object Copy

)pcopy [***libno***] ***ws*** [***:pass***] [$name_1$ $name_2$ ...]

This command is a variant of)copy, and is discussed under that heading, earlier in this chapter.

)port

Locate User

)port [***nnnnnnn*** | ***usr***]

This command helps you to locate the port and task number of another user. You may want the task number so that you can address a message with the)msg command.

Following the name of the command, you write *either* the user number, *or* the first three letters of the user name.

If you specify the user number and that user is signed on, SHARP APL reports all tasks running under that user number. If you specify the first three letters of the user name, SHARP APL reports that information for all users whose names start with those three letters and that are currently signed on.

The report SHARP APL displays shows (for as many tasks as match the number or letters you have supplied) the following three fields:

1. *Port number* The last three digits refer to individual ports. The high-order digits refer to the node in the communications network at which the port is located.
2. *First three letters of the user name* This short form is deliberately chosen to provide confirmation when you are already familiar with the full user name, without disclosing the full name when you are not.
3. *Task number* Only the task number is relevant in addressing a message with)msg or)msgn.

)reset

Clear SI Stack

)reset

This command is a synonym for)sic, described below.

)save

Save Active Workspace

)save **[*libno*]** **[*ws*]** **[*:pass*]**

This command causes SHARP APL to store a duplicate of the active workspace. It leaves the contents of the active workspace unchanged. When you save a workspace into a *public library*, the name of the workspace must be preceded by the library number of the public library.[3]

Used with no name following it, the command)save causes SHARP APL to save a duplicate of the active workspace under the name it acquired when you loaded it or the name that you later gave it by the)wsid command. That updates the saved workspace, replacing the former version with a duplicate of the present active workspace. However, you may resave a workspace only if you are its owner.

When you load a duplicate of a saved workspace, the active workspace has the same name as the saved workspace. The active workspace resulting from the command)clear has no name. You may give your active workspace a name by the command)wsid. In either case, the name of the active workspace is a *potential* name. That is, it is the name under which SHARP APL would save your active workspace *if* you were to execute the command)save. When the workspace as yet has no name, SHARP APL rejects the command)save with the message `not saved, this ws is clear ws.`

When you supply a workspace name following the word)save, SHARP APL saves a duplicate of the active workspace in your library (or in the public library that you indicate) under the name you state. However, if you try to assign a new name that duplicates the name of an existing saved workspace, it is rejected with the message `not saved, this ws is` *XXX*, where *XXX* is the existing name of the active workspace.

After the)save command is successfully carried out, the name under which you saved it also becomes the name of the active workspace.

Workspace Locks

When you save a workspace, you may at the same time *lock* it. A locked workspace has a *password* of up to eight letters and/or numerals. You lock a workspace by entering a colon and the proposed password at the end of the)save command. Once a saved workspace has been locked, anyone (including you) making any reference to the saved workspace or its contents must include a colon and the password in the command.

[3] When you save a workspace in a public library, you are its owner, and it counts as part of your quota of saved workspaces, even though it no longer appears in the list displayed by the)lib command for your own user number.

)seal
Seal Workspace

)seal

Sealing a workspace makes it impossible to disperse its contents or display the definitions of user-defined verbs stored in it when it was sealed. The command takes immediate effect on the active workspace, and remains in effect when the active workspace is saved. Sealing a workspace has the following effects:

- All user-defined verbs in the workspace when the)seal command is executed are locked.
- No user-defined verb whose definition is locked can be erased. Nor may a name referring to a locked verb be given a new referent (for example, as a consequence of □pdef, □fx, etc.).
- The command)copy into a sealed workspace is treated as if it were)pcopy; it cannot replace any existing global object, regardless of whether it was or was not in the workspace when the workspace was sealed.
- No locked function can be copied from a saved workspace that has been sealed.

The command)seal is effective only when the state indicator is empty. If a suspended function is on the state indicator, SHARP APL displays the state indicator and does not seal the workspace.

Watch out: Sealing a workspace is irreversible. You not only prevent others from copying or displaying verbs from your workspace, you also prevent yourself from making revisions to it. Somewhere else you should keep an unsealed version of your sealed workspace.

)sesm
Session Handler Commands

)sesm

This command is part of the session handler rather than being part of SHARP APL. See the manual for the session handler you are using.

)sic
State Indicator Clear

```
)sic
```

This command abandons execution of *all* suspended and pendent verbs, thereby clearing the state indicator. The referent of any local name is lost. The record of the last line entered from the keyboard, recallable with the command) or)***n***, is discarded.

The effect of this command is different from that produced by the naked branch →. The branch abandons execution of the verb most recently started from immediate execution, together with all verbs that it may have invoked, but has *no* effect on verbs whose execution started earlier than that. See also □*trap* action *d*.

The command)*reset* is a synonym for)*sic*.

)si
State Indicator

```
)si
```

)sinl
State Indicator with Namelist

```
)sinl
```

)siv
State Indicator with Namelist

```
)siv
```

This command causes SHARP APL to display a list of the verbs whose execution is pendent or suspended, and which are therefore on the execution stack (or *state indicator*). The list shows each level of execution on a separate line, showing the name of the verb and the number of the line that is currently active.

For a verb that is *suspended*, the display shows an asterisk. The line number in that case is the number of the line next to be executed when execution is resumed.

To illustrate, consider the following display:

```
      )si
get[4] *
verify[1]
mean[1] *
analyze[3]
report[2]
```

The verb *get* is suspended, and has not completed execution of line 4. If resumed, it would start at the beginning of line 4. The verb *get* was invoked by line 1 of *verify*. The verb *verify* is waiting for *get* to complete, at which point *verify* will resume automatically its work on the line indicated (line 1).

The verb *mean* is also suspended. Evidently, following that suspension, rather that resume its execution, you invoked *verify* (perhaps in an effort to examine the cause of the interruption in *mean*).

mean was invoked by a sentence on line 3 of *analyze*, which in turn was invoked by line 2 of *report*. *report* must have been invoked in a sentence entered from the keyboard during immediate execution.

The verb ⍎ or a request for input via ⍞ appears in the state indicator in the same way as a user-defined verb, but without a line number.[4]

A table having the same appearance as *)si* is returned as the result of the verb `2 ⎕ws 2`. A list of the line numbers that appear is returned by the verb *⎕lc* (for *line counter*).

Optionally, the command may include the letters *nl* (for *namelist*), so that it is written *)sinl*. In that case, the display is organized in the same way, but also includes on each line a list of names local to the verb on that line. The information about the use of names is returned in a different form by the verb `5 ⎕ws`.

The obsolescent form *)siv* (state indicator with variables) is still accepted as a synonym for *)sinl*.

4 For examples, see the section "Status of Work in Progress" in Chapter 9, "Control of Execution".

)symbols

Set Symbol Table Size

)symbols [*n*]

Used without an argument, this command reports the current size of the symbol table. The symbol table must contain an entry for every name in the workspace, including the name of every user-defined noun and verb. It also contains every name referred to in a definition that exists in the workspace.

Used with a numeric argument, this command causes the interpreter to set aside space in the workspace for a symbol table with the number of entries requested. (You may not request a table smaller than the number of names already in use; such a request is rejected with the message *symbol table full*.)

Since SHARP APL is able to increase the size of the symbol table automatically, it is not essential to do this. However, to avoid some inefficiencies in having to resize the table dynamically, it may be desirable to set the size large enough to accommodate all forseeable names. Conversely, when you know that few names are needed, you can make the symbol table smaller and thereby recover some of the space that would otherwise have been reserved for it.

The symbol table size and number of symbols in use are also reported by `2 ⎕ws 3`.

)terminal

Terminal Type

)terminal [*type*]

The command)terminal lets you tell SHARP APL which of several recognized terminals you are using. This will not fundamentally affect the output you receive, but it does permit SHARP APL's session handlers to make more accurate allowances for peculiarities of certain terminals.

Used with no argument,)terminal reports the current terminal type. The possible arguments are as follows:[5]

aj832	Synonym for *nulls*.
get1200	GE-Terminet lineprinter terminal at 1200 baud, without Xon/Xoff flow control.

[5] A current list can be obtained from the verb *terminals*, in public workspace `5 term`.

`idles`	Default for non-T-tasks, and for ASCII terminals at 300 baud or less.
`mrx1240`	Memorex 1240 lineprinter terminals.
`noidles`	Default for all terminals communicating asynchronously at 1200 baud or more. Assumes that the terminal does not require any line-ending idle characters, and that flow-control between terminal and host is handled with DC1–DC3 (also called XON/XOFF). This setting speeds up the transmission of data from SHARP APL, but may cause loss of characters if the terminal is not buffered, or lacks flow-control when its buffer becomes full.
`nulls`	Default for Telex terminals and for ASCII terminals with APL characters running asynchronously at 300 baud or less. Replaces idles by nulls, but produces the same number of them as `idles`. Synonymous with `aj832`.
`realtty`	Model 33 Teletype at 110 baud.
`sdata`	Scopedata terminal with *auto-LF* (that is, CR includes LF).

The terminal type can also be enquired or set using the verb `term` in workspace `1 wsfns`.

)vars
Display Noun Names

`)vars` [***firstname***]

This command causes SHARP APL to display a list of those names whose visible referent is to a noun (variable), in alphabetical order, in the same format as `)fns`. The names listed are the same as those returned by `⎕nl 2` or `1 ⎕ws 2`.

When the command is followed by a name (shown here as ***firstname***), the list shows only those names that occur in the alphabetized list at or after the position of ***firstname***.

)wsid

Set Workspace Name

)wsid [***newname***]

Used with no argument,)wsid displays the name of the active workspace. When the workspace comes from a library other than your own, the display includes the library number.

Used with an argument,)wsid assigns that name to the active workspace. The name of the active workspace is relevant only when you subsequently save the workspace. See the discussion accompanying the description of the command)save, above.

Watch out: If you set the name of the active workspace to the name of an already-saved workspace, you defeat SHARP APL's check against overwriting one of your saved workspaces.

)xload

Load Workspace without Autostart

)xload [***libno***] ***ws*** [***:pass***]

See the entry accompanying the description of)load.

18 *Messages*

When for some reason the SHARP APL system encounters a problem and cannot proceed, it may display a message to indicate what has happened. This chapter describes what the various messages mean, the circumstances that may have produced them, and in some cases what remedial action you may be able to take.

Sources of messages and trouble reports

Messages originate in different parts of the system. They can be roughly classified according to where they are emitted, as follows:

Communications network or transmission control Certain messages originate not in the APL system, but in the communications network or the transmission control devices. They are not sent by SHARP APL; in fact, they may mean that it is impossible to communicate with SHARP APL. Such messages are site-dependent.

The host operating system Because SHARP APL runs as one of many jobs under the general supervision of the MVS operating system (the *host*), there may be circumstances under which the host system, rather than APL, sends you a message. Such site-dependent messages are outside the scope of this manual.

Session handler The SHARP APL system's session handler is responsible for communication between a terminal and the SHARP APL interpreter. It constructs an input line from the signals it receives from your terminal. When it receives an unacceptable transmission, it may send a message indicating trouble. That entire sequence may take place without involving the SHARP APL interpreter or your workspace.

Trouble reports regarding system commands A trouble report describes a problem encountered in attempting to carry out a system command or involving the ∇-editor. Since those can be invoked only from an interactive task, the corresponding messages can occur only during a T- or an S-task. You cannot correct a command, once it has been entered, other than to re-enter it correctly.

Error or interrupt messages from the SHARP APL interpreter

Each of these messages is caused by an event detected by the SHARP APL interpreter. The event is either an error in a line being executed (either a line entered from the keyboard for immediate execution or a line within a user-defined verb) or an interrupt signaled from outside the workspace (usually from the keyboard) while a line is being executed. Some (but not all) errors or interrupts can be intercepted by an appropriate setting of `⎕trap`, and are said to be *trappable*.

Trappable Errors and Interrupts

Each time a trappable event occurs, the system sets a new value for the system noun `⎕er` (*event report*). If the event is *not* trapped, execution halts. If the task is an N-task or B-task, the task is terminated and its active workspace is saved. If the task is a T-task or an S-task, the system displays a message. The message is identical to the content of `⎕er`, except that `⎕er` also contains an event number while the display does not. Procedures for handling trappable events are discussed in Chapter 10, "Event Handling".

Following an untrapped error or interrupt, it may be possible to take corrective action. If the message is in response to an entry from the keyboard, you may be able to correct the entry and resubmit it. If the message is in response to a problem that arose during execution of a user-defined verb, you may be able to correct the situation and resume execution.

Form of an Error Message

When the APL interpreter encounters an untrapped error or interrupt, it sends you a three-part message. The first part is a general description of the type of problem (for example, `syntax error` or `value error`). Then it shows you the line containing the error. If the line is within a user-defined verb, it shows also the name of the verb and the number of the line on which it was working when it encountered the error. (When the verb's definition is locked, the system shows the verb's name and the line number, but not what is on the line.)

Below that, on the third line, the system displays a caret. The caret points to the part of the line on which the interpreter was working when it encountered the error. For some errors, the system can point quite precisely to what is wrong. For others, it is harder to point to what is wrong. For example, if you omitted a parenthesis, it is hard for the system to know where you

should have put it. Instead, it shows where it was working when it found that parentheses do not match.

How Much Has Been Executed

When the system displays a line of APL marked with a caret, the interpreter may have already executed part of the line, before it discovered the error. In general, you can tell what has been executed by the position of the caret. But it is prudent to check further; the caret's position is not a definitive guide. An approximate rule is:

- A `syntax error` due to mismatched quotes, or inclusion of illegal characters as part of an unquoted part of a line, is signaled before any part of the line is executed. The caret appears one character past the right-most end of the line.
- Within a *line* containing diamonds, statements to the right of the caret have *not* yet been executed.
- Within the *sentence* to which the caret is pointing, verbs to the right of the caret *have* been executed.

If the statements or verbs that have already been executed produce some lasting change (for example, by respecifying the value of a noun or of a file component), you may have to make allowance for that before you re-execute the line.

Point of View in Describing an Error

Most errors arise because you did not state what you wanted in a way acceptable to the interpreter. From your point of view, the most helpful message would be one telling you what you *should* have done. But what you should have done can be defined only by referring to what you have in mind. The computer has no way of knowing that. So whenever it encounters a problem, the system describes what was wrong *from its point of view*. The descriptions that follow describe what the various error messages mean to the system, and in some cases suggest what may have brought them about, and what corrective action you may be able to take.

National Language Translation: `)nlt`

The command `)nlt` permits you to determine in which language error messages appear.

The setting of `)nlt` affects all messages issued by the SHARP APL interpreter, but not messages that originate in the communications control.

A meaningful argument to `)nlt` is the name of an available language, spelled in that language as nearly as possible but without national use characters. SHARP APL at present recognizes the following:

```
)nlt english
)nlt deutsch
)nlt francais
```

Each installation sets its own default value for national language.

Error Messages By Event Number

This section lists the numbered "events" in order of event number. For each event, it shows the associated message in English, French, and German. In the alphabetical section that follows, all messages are listed in alphabetical order. However, a particular message is only described once. If it is a message that has an event number, the alphabetical entry refers you back here. If it is a message that has no event number, the alphabetical entry for the English version has the description, and the other entries tell you to look under the English entry.

1
`ws full`

Fr: `pas de place`
D: `ab voll`

If the system were to execute the verb, or to copy the requested objects, it would require more space than is available in the workspace. When you save a workspace containing a shared name, before carrying out the save, the system must record the current value of any shared name. If your partner has given them values so large that your workspace has no room to accommodate them, the system will report `ws full` and will not save the workspace unless your task has been forced off by a communications failure or a system crash.

Remedy: Sometimes it suffices to erase unwanted objects in the workspace. If the workspace contains suspended or pendent verbs, `)sic` or → may help. Alternatively, you may need to alter the algorithm so that its intermediate storage is not so extensive. (When this condition arises as the result of executing an APL expression, it is trappable as event 1. But `ws full` arising during `)load`, `)copy`, or `)save` is not trappable.)

2
`syntax error`

Fr: `erreur de syntaxe`
D: `syntaxfehler`

The syntax analyzer has detected a statement appearing to violate the syntactic rules of the language. This could be for any of several reasons; for instance:

- You have invoked a verb requiring a right argument without one.
- You have unbalanced parentheses, brackets, or quotes.
- You have juxtaposed two names referring to nouns with no indication of a verb to be applied to them.

3 Fr: *erreur d'indexation*
index error D: *indexfehler*

Within the brackets delimiting an index or axis, you have supplied a value that is not a valid index for the object being indexed. One possible cause of the problem is the wrong setting for the system noun □*io*.

4 Fr: *erreur de rang*
rank error D: *rangfehler*

You have invoked a monadic verb whose argument is limited to a particular rank with an argument of some other rank. Or you have executed a dyadic verb with arguments not corresponding in rank when they should.

5 Fr: *erreur de longueur*
length error D: *laengenfehler*

You invoked a dyadic verb requiring its arguments to match in length, but your arguments do not match along the required axes.

6 Fr: *objet inexistant*
value error D: *name ohne wert*

You have referred to what appears to be the name of a noun or verb with no visible value. This might be because the name is shadowed during execution of a verb, or because the value really does not exist in the workspace. This message often arises because you have misspelled the name of the noun or verb.

7 Fr: *erreur de format*
format error D: *formatfehler*

You have supplied an invalid argument to □*fmt*. The system displays the portion of the left argument of □*fmt* that it was attempting to interpret.

8 Fr: *resultat inexistant*
result error D: *resultat fehlt*

An expression produced no result (in a context in which a result is expected). ⍎'' and monadic ⊣ produce no result; neither does a user-defined verb whose header does not include a name for the result, or whose execution ends without ever assigning a value to the result.

9
si error

Fr: *erreur de pile*
D: *si fehler*

You have changed the definition of a suspended verb in a way that changes the use of names so that SHARP APL cannot find the definition of a pendent verb.

Remedy: Clear the state indicator by executing → as needed, or clear the state indicator completely by the command)*sic*.

11
domain error

Fr: *erreur de domaine*
D: *unzulaessiges argument*

You have invoked a verb, but the arguments you supplied to it are outside the verb's domain. Common examples: you have tried to divide by zero, or to catenate numbers and characters, or you have supplied a fractional argument to a verb defined only on integers. Consult the description of the verb in question for the criteria for an acceptable argument.

15
symbol table full

Fr: *debordement de table de symboles*
D: *symboltabelle voll*

The line you are trying to execute, or the package whose contents you are trying to define, or the object you are trying to copy, requires the use of names for which there is no space in the symbol table, and the symbol table can no longer be automatically resized. This can only occur if the symbol table is already at its maximum size.

Remedy: Correction will require resizing the workspace or reorganizing the application so that either a larger symbol table is possible or the need for so many distinct symbols is reduced.

16
nonce error

Fr: *enonce non defini*
D: *nicht implementiert*

The expression you entered appears valid, but SHARP APL is not presently (that is, for the nonce) equipped to execute it. This message may be encountered if you execute a proposed verb that the system recognizes but whose implementation is not yet released.

18
file tie error

Fr: *fichier: lien invalide*
D: *datei bindefehler*

You have specified a tie number as if it were tied when in fact it is not, or as if it were free to be tied when in fact it is already tied.

19
file access error

Fr: *fichier: acces invalide*
D: *datei zugriffsfehler*

You have tried to tie a file to which you do not have access, or to tie or use it without the correct passnumber, or to use with it a verb for which the file's access matrix does not give you authorization. The system keeps a log of file access errors.

20
file index error

Fr: *fichier: erreur d'indexation*
D: *datei indexfehler*

You have invoked □*read*, □*replace*, □*rdci*, or □*drop* with an argument referring to a nonexistent file component, or you have attempted to append a component number greater than 214748647 to a file.

21
file full

Fr: *fichier: pas de place*
D: *datei voll*

You have attempted to append a new component to a file whose current size already exceeds the number of bytes reserved for that file.

Remedy: Resize the file. See the description of □*resize* in Chapter 16, "System Nouns and Verbs".

Note: You might expect that replacing some of the existing components with data requiring less space would decrease the total storage of the file. While that is true in the long run, it does not always reduce the total size immediately. That is because the physical storage freed by doing that may not be available to the new components you want to append until after the operations staff has performed periodic file maintenance.

22
file name error

Fr: *fichier: nom invalide*
D: *datei namensfehler*

The left argument of □*tie*, □*stie*, or □*create* is not a valid name, or, where the verb must refer to an existing file, not the name of an existing file. In the case of □*create*, the file name may already exist.

23
file damaged

Fr: *fichier endommage*
D: *datei beschaedigt*

The file you are trying to use has been marked *damaged* in the directory the system maintains. A file is marked damaged if the file crash recovery program cannot successfully recover the file after a system crash. In addition, a system programmer can mark a file as damaged if required. While the file is damaged, the only verbs you can use with it are □*tie*, □*stie*, □*erase*, and □*untie*. Any other verb is rejected with the message *file damaged*. A damaged file is not archived, and will be lost following a

"full dump/full restore" (part of the SHARP APL file system's periodic maintenance). You can ask to have the previous version of the file restored from archive. Notify the system administrators for assistance.

24
file tied

Fr: *fichier deja relie*
D: *datei gebunden*

You have attempted to tie a file you have already tied or (on the mainframe) a file exclusively tied by another task. If the file is tied by another task, there is no immediate remedy, other than to wait. You may want to arrange with other users of the file to avoid exclusive ties, or amend the access matrix to prohibit them. With □*erase*, another task may have the file share tied.

25
file system hardware error

Fr: *fichier: ennui materiel*
D: *datei system hardware fehler*

The system is unable to read correctly from the physical device storing the component you have asked to read or write. Report the occurrence to the system administrators so that corrective action can be taken.

26
file system error

Fr: *fichier: ennui systeme*
D: *datei system fehler*

Something is seriously wrong with the file system. Report the occurrence to the system administrators so that corrective action can be taken.

28
file system not available

Fr: *fichier: systeme non disponible*
D: *datei system nicht verfuegbar*

You have invoked a file verb other than □*avail* at a time when, although APL is running, the file system is not. Report the occurrence to the system administrators so that corrective action can be taken.

Remedy: Wait a few minutes and try again.

29
file library not available

Fr: *fichier: bibliotheque non disponible*
D: *datei bibliothek nicht verfuegbar*

The system's physical storage of files is divided into segments called *volume classes*. The volume class in which a file is stored depends on the number of the library to which it belongs. The volume class containing the file you have requested is at present unavailable. That is usually because of periodic maintenance, or perhaps because of trouble with the physical devices on which that volume class is stored. The problem may be transitory, but if

it persists the system administrators may be able to advise you when the volume class (and hence that file library) will be available.

30
`file system ties used up`

Fr: `fichier: systeme surcharge`
D: `datei system bindelimit erreicht`

The system now has tied the maximum number of distinct files. (The number depends on the installation.) The situation may be temporary. Bring the occurrence to the attention of the system administrators so that steps can be taken to decrease the likelihood of a recurrence.

31
`file tie quota used up`

Fr: `fichier: quota de liens epuise`
D: `dateien bindelimit erreicht`

The system administrators set for each SHARP APL system a maximum on the number of files that one task may tie at one time. You have attempted to tie a file in excess of that limit.

Remedy: Untie files that you are not currently using.

32
`file quota used up`

Fr: `fichier: quota epuise`
D: `dateien limit erreicht`

You have requested creation of a new file, but the number of files you own is already equal to your quota.

Remedy: Erase an unwanted file, or establish a larger quota. You establish a larger quota by requesting it from the system administrators.

33
`file reservation error`

Fr: `fichier: quota d'espace epuise`
D: `dateien speicherlimit erreicht`

You have requested space for a file (either by resizing an existing file or by creating a new one) in a way that would make the sum of the reservations for all your files greater than your file reservation quota.

Remedy: First erase or resize some of your existing files, and/or ask the system administrators for an increased file reservation limit.

34
`file system no space`

Fr: `fichier: pas de place sur le systeme`
D: `datei system kein platz`

You have requested to append components to a file, in a way that is within your limits, but which the file system is unable to honor because it has insufficient space. Notify the system administrators so that corrective action can be taken.

38
file component damaged

Fr: *fichier: composante endommagee*
D: *datei komponente beschaedigt*

You have attempted to read a file component that is unreadable because of an inconsistent file system internal directory following a system failure. There is no immediate remedy. Notify the system administrators.

45
ws locked

Fr: *espace verrouille*
D: *ab gesperrt*

While referring to a workspace, the password you have supplied does not match the password with which the workspace was saved. This could arise either because the workspace requires a password but you did not supply the right one, or because it has no password and you did supply one. If your reference to the workspace occurred in an APL expression (that is, one invoking □*load* or □*qload*), this event is trappable as event 45. But if your reference was in a system command, it is not trappable. (When this happens following a command)*load* or)*copy*, □*er* is not set. But when it happens in response to □*load* or □*qload*, □*er* is set to show event 45.)

46
ws not found

Fr: *espace inexistant*
D: *ab nicht gefunden*

The workspace you requested cannot be found in the library stated (or implied) in your request to load or copy a workspace. If your reference to the workspace occurred in an APL expression (that is, one invoking □*load* or □*qload*), this event is trappable as event 46. But if your reference was in a system command, it is not trappable. (When this happens following a command)*load* or)*copy*, □*er* is not set. But when it happens in response to □*load* or □*qload*, □*er* is set to show event 46.)

47
ws not readable

Fr: *espace illisible*
D: *ab nicht lesbar*

The system is unable to read the saved workspace you have requested to load or copy. No immediate recovery is likely. Contact the system administrators, who will attempt to recover the workspace, or failing that, to restore it from the backup tapes of saved workspaces. (When this happens following a command)*load* or)*copy*, □*er* is not set. But when it happens in response to □*load* or □*qload*, □*er* is set to show event 47, which is trappable.)

67
swap error, clear ws

Fr: *erreur d'echange, espace vide*
D: *swap fehler, ab leer*

The system finds (in the course of *swapping* your active workspace from auxiliary storage) that it is no longer able to read your active workspace.

It is not possible to recover the active workspace following a swap error. After a swap error, the preceding values of the session nouns □*sp* and □*ht* are lost, and in the ensuing clear workspace, each is an empty list. Notify the system administrators of the problem, so the malfunction can be fixed. If this error occurs during an N-task or B-task, the task is terminated, and □*er* in its saved workspace indicates event 67; however, event 67 is not trappable.

68
system error, clear ws

Fr: *erreur systeme, espace vide*
D: *system fehler, ab leer*

The system has encountered an undiagnosed error while executing a command or an APL statement in your workspace. Because a damaged workspace is a potential security hazard, the system clears your workspace. Following a system error, the former values of the session nouns □*sp* and □*ht* are lost, and in the ensuing clear workspace, each is an empty vector.

To help the SHARP APL system personnel in locating the source of the trouble, it will be useful if you keep a record as exact as possible of what you were doing from the preceding)*load* or)*clear* to the moment the system error occurred. This event is not trappable. The system clear the workspace, then sets □*er* to report event 68.

72
interface quota exhausted

Fr: *vp: quota epuise*
D: *gv limit erreicht*

You have attempted to offer to share more names at one time than the SHARP APL system allows. The number is established by the system administrators for each installation.

Remedy: Retract some shares.

73
no shares

Fr: *vp: non operationnel*
D: *gv prozessor nicht aktiv*

You have invoked a shared name verb when the system's Shared Variable Processor is not running. This is an unusual state of affairs. It may be sufficient to wait briefly. Otherwise, consult the system administrators.

74
interface capacity exceeded

Fr: *vp: interface surchargee*
D: *gv kapazitaet ueberschritten*

You have attempted to assign to a shared variable a value requiring more storage space than the system's Shared Variable Processor can accept.

75
share table full

Fr: *vp: trop d'offres sur le systeme*
D: *gv tabelle voll*

The Shared Variable Processor's internal table of names shared between tasks is full. The problem may be transitory (because a temporary surge in activity will subside and eventually will free entries in the table). If the problem is chronic, it will be necessary for the system administrators to arrange for a larger share table.

76
processor table full

Fr: *vp: trop d'utilisateurs*
D: *prozessoren tabelle voll*

You have invoked □*svn*, □*svq*, or dyadic □*svo*, which requires the Shared Variable Processor to create for you an entry in its table of processor ID's, but the table has no room for a further entry.

This should be a transitory problem, so it may be sufficient to wait briefly and try again. Otherwise, consult the system administrators.

77
identification in use

Fr: *vp: identification deja utilisee*
D: *kennung bereits in verwendung*

You have attempted to use □*svq* or dyadic □*svo* at a time when another task is using shared names, and already has established the clone ID that you are now using.

Remedy: Set a unique clone ID, using □*svn*. See Chapter 11, "Shared Variables".

System Nouns with Invalid Values

The following events arise when the indicated system noun has been given an unacceptable value. (See the description of the noun in question in Chapter 16, "System Nouns and Verbs".)

78
⎕pp error

Fr: *⎕pp erronne*
D: *⎕pp fehler*

79
⎕pw error

Fr: *⎕pw erronne*
D: *⎕pw fehler*

80
⎕io error

Fr: *⎕io erronne*
D: *⎕io fehler*

81
⎕rl error

Fr: *⎕rl erronne*
D: *⎕rl fehler*

82
⎕ct error

Fr: *⎕ct erronne*
D: *⎕ct fehler*

83
⎕ht error

Fr: *⎕ht erronne*
D: *⎕ht fehler*

84
⎕ps error

Fr: *⎕ps erronne*
D: *⎕ps fehler*

85
⎕fc error

Fr: *⎕fc erronne*
D: *⎕fc fehler*

95
ws not writable

Fr: *systeme incapable d'ecrire cet espace*
D: *ab nicht speicherbar*

You have attempted to *)save* a workspace, but a hardware failure has prevented it from completing successfully. Consult the system administrators.

Remedy: There is no immediate remedy.

Interrupt Events

1001
`stop`

Fr: `stop`
D: `stop`

While executing a user-defined verb, SHARP APL has encountered a line for which a *stop* has been set (see "Stop and Trace Controls" in Chapter 9, "Control of Execution").

1002
`attention`

Fr: `attention`
D: `achtung`

While executing a user-defined verb, SHARP APL has a received a *weak interrupt*. Execution halts at the beginning of the next line to be executed.

1003
`interrupt`

Fr: `interruption`
D: `unterbrechung`

The system has received a *strong interrupt* signal. It has cut short execution of the line it was executing.

When a verb is executing, a strong interrupt is signaled by two successive strikes of *Break*, *PA1*, or *Attention*. During input, the means by which an input interrupt is signaled varies with the type of terminal; see "Escape from ⍞-input and ⎕-input" in Chapter 4, "Nouns and Pronouns".

Watch out: Part of the line may have been executed. An instruction to store data occurring within this statement and to the right of the caret may have been carried out already. That includes assigning a new value to a noun or storing something in a file. If you consider re-executing the line, notice that that storage (noun or file component) may already have received a new value, and the line may not be trivially re-executable.

The system discriminates several different interrupts, according to the interrupted activity:

1004 Interrupt during ⎕- or ⍞-input.

1005 Interrupt while awaiting a shared name access.

1006 Interrupt during a file operation.

1007 Interrupt during file backup.

1003 An interrupt during execution, but not 1005, 1006, or 1007.

1004
input interrupt

Fr: *interruption en entree*
D: *eingabe unterbrechung*

You have interrupted input from the keyboard (in response to □ or ⍞). The means by which an input interrupt is signaled varies with the type of terminal; see "Escape from ⍞-input and □-input" in Chapter 4, "Nouns and Pronouns".

1005
sv interrupt

Fr: *vp: interruption*
D: *gv unterbrechung*

You have signaled a *strong interrupt* (by two successive strikes of *Attention*, *PA1*, or *Break*) and have interrupted a set or a use of an interlocked shared name.

Remedy: Verify access control (□*svc*) and shared variable state (□*svs*). If it is appropriate to retry the access, re-execute the line. Trying the same access again does not constitute a new reference to the shared name, but rather a continuation of the previous attempt. See the caution on restarting described in event 1003.

1006
file interrupt

Fr: *fichier: interruption*
D: *dateien unterbrechung*

You have signaled an interrupt (two successive strikes of *Attention*, *PA1*, or *Break*) and thereby have interrupted a file operation.

Remedy: When ready, retry the operation.

Watch out: If the interrupted file operation is embedded in a long line, the system may have executed other parts of the line, making it imprudent to re-execute the line from the beginning. See the Caution on restarting described in event 1003.

1007
file backup interrupt

Fr: *fichier: interruption pendant archivage*
D: *dateien sicherungs unterbechung*

You have signaled an interrupt (two successive strikes of *Attention*, *PA1*, or *Break*) and have interrupted a file operation that is delayed during a system file backup. File backups do not usually take long, so the usual remedy is to try again in a few minutes. See the caution on restarting described in event 1003.

Alphabetic List of Messages

This section includes messages in English, German, and French. For messages associated with an event number, the entry refers you to the description in the numeric list earlier in this chapter. For messages not related to an event number, the entry refers you to the description accompanying the English version of the message.

ab leer

E: *clear ws*
Fr: *espace vide*

For a description of this message, see the entry under the English-language version, *clear ws*.

ab limit erreicht

E: *ws quota used up*
Fr: *quota epuise*

For a description of this message, see the entry under the English-language version, *ws quota used up*.

ab gesperrt

E: *ws locked*
Fr: *espace verrouille*

For a description of this message, see the entry in the preceding section under event 45.

ab nicht gefunden

E: *ws not found*
Fr: *espace inexistant*

For a description of this message, see the entry in the preceding section under event 46.

ab nicht lesbar

E: *ws not readable*
Fr: *espace illisible*

For a description of this message, see the entry in the preceding section under event 47.

ab nicht speicherbar

E: *ws not writable*
Fr: *systeme incapable d'ecrire cet espace*

For a description of this message, see the entry in the preceding section under event 95.

ab voll

E: *ws full*
Fr: *pas de place*

For a description of this message, see the entry in the preceding section under event 1.

abgemeldet

E: *signed off*
Fr: *usager absent*

For a description of this message, see the entry under the English-language version, *signed off*.

achtung

E: *attention*
Fr: *attention*

For a description of this message, see the entry under the English-language version, *attention*.

anfrage verloren

E: *message lost*
Fr: *message perdu*

For a description of this message, see the entry under the English-language version, *message lost*.

apl down/xxx

Fr: *apl down/xxx*
D: *apl down/xxx*

The SHARP APL system is not running. This message is usually transitory; the operations staff will be aware of the condition and will attempt corrective action. The letters after the slashes are intended to help the staff diagnose the problem and need not concern you.

After you receive this message, your link to the system remains established and any further entries are interpreted as attempts to sign on to the same SHARP APL system.

apl full/xxx

Fr: *apl full/xxx*
D: *apl full/xxx*

The network node required to connect you to the SHARP APL system at present has insufficient capacity to do so. The letters after the slashes are intended to help the staff diagnose the problem and need not concern you.

apl not available

Fr: *apl not available*
D: *apl not available*

The operations staff is in the process of starting SHARP APL, but SHARP APL is not yet ready to accept sign-ons. This condition should not last more than a few minutes.

apl reset/xxx

Fr: *apl reset/xxx*
D: *apl reset/xxx*

A transmission controller has detected a reset signal from the host computer, implying that the previous APL session probably ended in a system crash. The letters after the slashes are intended to help the staff diagnose the problem and need not concern you.

After you receive this message, if all goes well, depending on what caused the system crash, it may be possible to restart the system. As a step in restarting, the system will first attempt to recover all workspaces as they were, or, failing that, as they were several seconds before the crash occurred.

attention

Fr: *achtung*
D: *attention*

This message is not displayed. However, it is the error description appearing in □*er* when execution is halted by a single strike of *Attention*, *PA1*, or *Break*. Execution halts after completion of work on the current line. If *attention* occurred during work on a user-defined verb, □*er* (and □*lc*) shows the number of the line next to be executed. See the description of event 1002.

autorisation retiree...

E: *number locked out...*
D: *benutznummer gesperrt...*

For a description of this message, see the entry under the English-language version, *number locked out...* .

benutzernummer existiert nicht...

E: *number not in...*
Fr: *numero inconnu...*

For a description of this message, see the entry under the English-language version, *number not in...* .

benutzernummer gesperrt...

E: *number locked out...*
Fr: *autorisation retiree...*

For a description of this message, see the entry under the English-language version, *number locked out...* .

benutzernummer in gebrauch

E: *number in use*
Fr: *numero actif*

For a description of this message, see the entry under the English-language version, *number in use*.

bibliothekstabelle voll

E: *library table full*
Fr: *debordement de table de bibliotheque*

For a description of this message, see the entry under the English-language version, *library table full*.

bibliotheque inexistante

E: *improper library reference*
D: *unerlaubter bibliothekszugriff*

For a description of this message, see the entry under the English-language version, *improper library reference*.

c'est...

E: *is...*
D: *ist...*

For a description of this message, see the entry under the English-language version, *is...* .

c'etait...

E: *was...*
D: *war...*

For a description of this message, see the entry under the English-language version, *was...* .

clear ws

Fr: *espace vide*
D: *ab leer*

Following an interrupt in a workspace in which *clearout* has been set, the system in effect executes the command *)clear* on your behalf. This event is not trappable.

commande invalide

E: *incorrect command*
D: *ungueltige systemanweisung*

For a description of this message, see the entry under the English-language version, *incorrect command*.

copie du...

E: *saved...*
D: *gespeichert...*

For a description of this message, see the entry under the English-language version, *saved...* .

datei beschaedigt

E: *file damaged*
Fr: *fichier endommage*

For a description of this message, see the entry in the preceding section under event 23.

datei bibliothek nicht verfuegbar

E: *file library not available*
Fr: *fichier: bibliotheque non disponible*

For a description of this message, see the entry in the preceding section under event 29.

datei bindefehler

E: *file tie error*
Fr: *fichier: lien invalide*

For a description of this message, see the entry in the preceding section under event 18.

datei gebunden

E: *file tied*
Fr: *fichier deja relie*

For a description of this message, see the entry in the preceding section under event 24.

datei indexfehler

E: *file index error*
Fr: *fichier: erreur d'indexation*

For a description of this message, see the entry in the preceding section under event 20.

datei komponente beschaedigt

E: *file component damaged*
Fr: *fichier: composante endommagee*

For a description of this message, see the entry in the preceding section under event 38.

datei namensfehler

E: *file name error*
Fr: *fichier: nom invalide*

For a description of this message, see the entry in the preceding section under event 22.

datei system bindelimit erreicht

E: *file system ties used up*
Fr: *fichier: systeme surcharge*

For a description of this message, see the entry in the preceding section under event 30.

datei system fehler

E: *file system error*
Fr: *fichier: ennui systeme*

For a description of this message, see the entry in the preceding section under event 26.

datei system hardware fehler

E: *file system hardware error*
Fr: *fichier: ennui materiel*

For a description of this message, see the entry in the preceding section under event 25.

datei system kein platz

E: *file system no space*
Fr: *fichier: pas de place sur le systeme*

For a description of this message, see the entry in the preceding section under event 34.

datei system nicht verfuegbar

E: *file system not available*
Fr: *fichier: systeme non disponible*

For a description of this message, see the entry in the preceding section under event 28.

datei voll

E: *file full*
Fr: *fichier: pas de place*

For a description of this message, see the entry in the preceding section under event 21.

datei zugriffsfehler

E: *file access error*
Fr: *fichier: acces invalide*

For a description of this message, see the entry in the preceding section under event 19.

dateien bindelimit erreicht

E: *file tie quota used up*
Fr: *fichier: quota de liens epuise*

For a description of this message, see the entry in the preceding section under event 31.

dateien limit erreicht

E: *file quota used up*
Fr: *fichier: quota epuise*

For a description of this message, see the entry in the preceding section under event 32.

dateien sicherungs unterbrechung

E: *file backup interrupt*
Fr: *fichier: interruption pendant archivage*

For a description of this message, see the entry in the preceding section under event 1007.

dateien speicherlimit erreicht

E: *file reservation error*
Fr: *fichier: quota d'espace epuis*

For a description of this message, see the entry in the preceding section under event 33.

dateien unterbrechung

E: *file interrupt*
Fr: *fichier: interruption*

For a description of this message, see the entry in the preceding section under event 1006.

debordement de table de bibliotheque

E: *library table full*
D: *bibliothekstabelle voll*

For a description of this message, see the entry under the English-language version, *library table full*.

debordement de table de symboles

E: *symbol table full*
D: *symbol tabelle voll*

For a description of this message, see the entry in the preceding section under event 15.

definitionsfehler

E: *defn error*
Fr: *erreur de definition*

For a description of this message, see the entry under the English-language version, *defn error*.

defn error

Fr: *erreur de definition*
D: *definitionsfehler*

You have invoked the ∇-editor improperly. The impropriety might be any of the following:

- While in immediate-execution mode, you have entered a line starting with [(which thus looks like a line number), or you have entered a line that starts with ∇ (which thus looks like a request to enter definition mode), but you have not provided a valid header. Following the initial ∇, you need:

- For an already existing verb, just the name of the verb.
- For a new verb, the name cannot already be in global use as the name of a noun or group.
- For a new verb (but not for an existing verb), you should write a complete header, including names for the result, arguments, and local names.

- While in execution mode, you have tried to open a verb's definition but the definition is locked.
- While editing a verb's definition, you have supplied a line number starting with `[` but not followed by a plausible line number and a closing `]`.
- While in verb-definition mode, you have tried to execute a system command other than `)`***nn***.
- You have tried to enter definition mode while answering a request for evaluated input.

defn error is not trappable (but note that an error in the use of ⎕*fd* or ⎕*fx* may be trappable as a *domain error*, or by analyzing the result of ⎕*fx*).

domain error

Fr: *erreur de domaine*
D: *unzulaessiges argument*

For a description of this message, see the entry in the preceding section under event 11.

eingabe unterbrechung

E: *input interrupt*
Fr: *interruption en entree*

For a description of this message, see the entry in the preceding section under event 1004.

eingabe zu lang

E: *line too long*
Fr: *ligne trop longue*

For a description of this message, see the entry under the English-language version, *line too long*.

enonce non defini

E: *nonce error*
D: *nicht implementiert*

For a description of this message, see the entry in the preceding section under event 16.

entry error

Fr: *erreur en entree*
D: *unzulaessiges zeichen in der eingabe*

The input processor has detected an overstrike that does not form a valid APL character. It echoes to you the characters it has received up to the invalid character, and waits for you to retransmit the remainder. Note that this error is detected before your entry reaches the SHARP APL interpreter, and cannot be trapped. The entire line can be discarded by entering an input interrupt sequence. See Chapter 4, "Nouns and Pronouns".

erreur de definition

E: *defn error*
D: *definitionsfehler*

For a description of this message, see the entry under the English-language version, *defn error*.

erreur d'echange, espace vide

E: *swap error, clear ws*
D: *swap fehler, ab leer*

For a description of this message, see the entry in the preceding section under event 67.

erreur d'indexation

E: *index error*
D: *indexfehler*

For a description of this message, see the entry in the preceding section under event 3.

erreur de domaine

E: *domain error*
D: *unzulaessiges argument*

For a description of this message, see the entry in the preceding section under event 11.

erreur de format

E: *format error*
D: *formatfehler*

For a description of this message, see the entry in the preceding section under event 7.

erreur de longueur

E: *length error*
D: *laengenfehler*

For a description of this message, see the entry in the preceding section under event 5.

erreur de pile

E: *si error*
D: *si fehler*

For a description of this message, see the entry in the preceding section under event 9.

erreur de rang

E: *rank error*
D: *rangfehler*

For a description of this message, see the entry in the preceding section under event 4.

erreur de syntaxe

E: *syntax error*
D: *syntaxfehler*

For a description of this message, see the entry in the preceding section under event 2.

erreur en entree

E: *entry error*
D: *unzulaessiges zeichen in der eingabe*

For a description of this message, see the entry under the English-language version, *entry error*.

erreur systeme, espace vide

E: *system error, clear ws*
D: *system fehler, ab leer*

For a description of this message, see the entry in the preceding section under event 68.

espace illisible

E: *ws not readable*
D: *ab nicht lesbar*

For a description of this message, see the entry in the preceding section under event 47.

espace inexistant

E: *ws not found*
D: *ab nicht gefunden*

For a description of this message, see the entry in the preceding section under event 46.

espace verrouille

E: *ws locked*
D: *ab gesperrt*

For a description of this message, see the entry in the preceding section under event 45.

espace vide

E: *clear ws*
D: *ab leer*

For a description of this message, see the entry under the English-language version, *clear ws*.

fehlerhafte anmeldung:

E: *incorrect signon:*
Fr: *syntaxe de liaison invalide:*

For a description of this message, see the entry under the English-language version, *incorrect signon:*.

fichier: acces invalide

E: *file access error*
D: *datei zugriffsfehler*

For a description of this message, see the entry in the preceding section under event 19.

fichier: bibliotheque non disponible

E: *file library not available*
D: *datei bibliothek nicht verfuegbar*

For a description of this message, see the entry in the preceding section under event 29.

fichier: composante endommagee

E: *file component damaged*
D: *datei komponente beschaedigt*

For a description of this message, see the entry in the preceding section under event 38.

fichier deja relie

E: *file tied*
D: *datei gebunden*

For a description of this message, see the entry in the preceding section under event 24.

fichier endommage

E: *file damaged*
D: *datei beschaedigt*

For a description of this message, see the entry in the preceding section under event 23.

fichier: ennui materiel

E: *file system hardware error*
D: *datei system hardware fehler*

For a description of this message, see the entry in the preceding section under event 25.

fichier: ennui systeme

E: *file system error*
D: *datei system fehler*

For a description of this message, see the entry in the preceding section under event 26.

fichier: erreur d'indexation

E: *file index error*
D: *datei indexfehler*

For a description of this message, see the entry in the preceding section under event 20.

fichier: interruption

E: *file interrupt*
D: *dateien unterbrechung*

For a description of this message, see the entry in the preceding section under event 1006.

fichier: interruption pendant archivage

E: *file backup interrupt*
D: *dateien sicherungs unterbrechung*

For a description of this message, see the entry in the preceding section under event 1007.

fichier: lien invalide

E: *file tie error*
D: *datei bindefehler*

For a description of this message, see the entry in the preceding section under event 18.

fichier: nom invalide

E: *file name error*
D: *datei namensfehler*

For a description of this message, see the entry in the preceding section under event 22.

fichier: pas de place

E: *file full*
D: *datei voll*

For a description of this message, see the entry in the preceding section under event 21.

fichier: pas de place sur le systeme

E: *file system no space*
D: *datei system kein platz*

For a description of this message, see the entry in the preceding section under event 34.

fichier: quota d'espace epuis

E: *file reservation error*
D: *dateien speicherlimit erreicht*

For a description of this message, see the entry in the preceding section under event 33.

fichier: quota de liens epuise

E: *file tie quota used up*
D: *dateien bindelimit erreicht*

For a description of this message, see the entry in the preceding section under event 31.

fichier: quota epuise

E: *file quota used up*
D: *dateien limit erreicht*

For a description of this message, see the entry in the preceding section under event 32.

fichier: systeme non disponible

E: *file system not available*
D: *datei system nicht verfuegbar*

For a description of this message, see the entry in the preceding section under event 28.

fichier: systeme surcharge

E: *file system ties used up*
D: *datei system bindelimit erreicht*

For a description of this message, see the entry in the preceding section under event 30.

file access error

Fr: *fichier: acces invalide*
D: *datei zugriffsfehler*

For a description of this message, see the entry in the preceding section under event 19.

file backup interrupt

Fr: *fichier: interruption pendant archivage*
D: *dateien sicherungs unterbrechung*

For a description of this message, see the entry in the preceding section under event 1007.

file component damaged

Fr: *fichier: composante endommagee*
D: *datei komponente beschaedigt*

For a description of this message, see the entry in the preceding section under event 38.

file damaged

Fr: *fichier endommagee*
D: *datei beschaedigt*

For a description of this message, see the entry in the preceding section under event 23.

file full

Fr: *fichier: pas de place*
D: *datei voll*

For a description of this message, see the entry in the preceding section under event 21.

file index error

Fr: *fichier: erreur d'indexation*
D: *datei indexfehler*

For a description of this message, see the entry in the preceding section under event 20.

file interrupt

Fr: *fichier: interruption*
D: *dateien unterbrechung*

For a description of this message, see the entry in the preceding section under event 1006.

file library not available

Fr: *fichier: bibliotheque non disponible*
D: *datei bibliothek nicht verfuegbar*

For a description of this message, see the entry in the preceding section under event 29.

file name error

Fr: *fichier: nom invalide*
D: *datei namensfehler*

For a description of this message, see the entry in the preceding section under event 22.

file quota used up

Fr: *fichier: quota epuise*
D: *dateien limit erreicht*

For a description of this message, see the entry in the preceding section under event 32.

file reservation error

Fr: *fichier: quota d'espace epuise*
D: *dateien speicherlimit erreicht*

For a description of this message, see the entry in the preceding section under event 33.

file system error

Fr: *fichier: ennui systeme*
D: *datei system fehler*

For a description of this message, see the entry in the preceding section under event 26.

file system hardware error

Fr: *fichier: ennui materiel*
D: *datei system hardware fehler*

For a description of this message, see the entry in the preceding section under event 25.

file system no space

Fr: *fichier: pas de place sur le systeme*
D: *datei system kein platz*

For a description of this message, see the entry in the preceding section under event 34.

file system not available

Fr: *fichier: systeme non disponible*
D: *datei system nicht verfuegbar*

For a description of this message, see the entry in the preceding section under event 28.

file system ties used up

Fr: *fichier: systeme surcharge*
D: *datei system bindelimit erreicht*

For a description of this message, see the entry in the preceding section under event 30.

file tie error

Fr: *fichier: lien invalide*
D: *datei bindefehler*

For a description of this message, see the entry in the preceding section under event 18.

file tie quota used up

Fr: *fichier: quota de liens epuise*
D: *dateien bindelimit erreicht*

For a description of this message, see the entry in the preceding section under event 31.

file tied

Fr: *fichier deja relie*
D: *datei gebunden*

For a description of this message, see the entry in the preceding section under event 24.

format error

Fr: *erreur de format*
D: *formatfehler*

For a description of this message, see the entry in the preceding section under event 7.

formatfehler

E: *format error*
Fr: *erreur de format*

For a description of this message, see the entry in the preceding section under event 7.

gesendet

E: *sent*
Fr: *transmis*

For a description of this message, see the entry under the English-language version, *sent*.

gespeichert...

E: *saved...*
Fr: *copie du...*

For a description of this message, see the entry under the English-language version, *saved...* .

gv kapazitaet ueberschritten

E: *interface capacity exceeded*
Fr: *vp: interface surchargee*

For a description of this message, see the entry in the preceding section under event 74.

gv limit erreicht

E: *interface quota exhausted*
Fr: *vp: quota epuise*

For a description of this message, see the entry in the preceding section under event 72.

gv prozessor nicht aktiv

E: *no shares*
Fr: *vp: non operationnel*

For a description of this message, see the entry in the preceding section under event 73.

gv tabelle voll

E: *share table full*
Fr: *vp: trop d'offres sur le systeme*

For a description of this message, see the entry in the preceding section under event 75.

gv unterbrechung

E: *sv interrupt*
Fr: *vp: interruption*

For a description of this message, see the entry in the preceding section under event 1005.

identification in use

Fr: *vp: identification deja utilisee*
D: *kennung bereits in verwendung*

For a description of this message, see the entry in the preceding section under event 77.

improper library reference

Fr: *bibliotheque inexistante*
D: *unerlaubter bibliothekszugriff*

You have invoked the command *)lib*, *)save*, or *)xload* with an improper argument. You may have attempted to invoke the command *)lib* to list the contents of someone else's private library, or to invoke *)save* to save into a private library other than your own, or to resave a public library workspace of which you are not the owner. You will also receive this message if you attempt to *)xload* a public workspace which do you not own or a workspace from someone else's library. This event is not trappable.

incommunicado

Fr: *non receptif*
D: *nicht ansprechbar*

You have entered the system command *)msg* or *)msgn* to address a message to a task unable to receive a message, either because it is not an interactive task (T or S) or because its owner has executed *)keyb nomsg*. This event is not trappable.

incorrect command

Fr: *commande invalide*
D: *ungueltige systemanweisung*

You have typed an entry beginning with a parenthesis but the system does not recognize it as an SHARP APL system command, or you have typed a command in a way that violates the command's syntax. This event is not trappable.

incorrect signon:

Fr: *syntaxe de liaison invalide:*
D: *fehlerhafte anmeldung:*

The first transmission the system has received from you contains characters that are not permitted in a valid sign-on. This may be simply because you

mistyped something. It may also arise from an inappropriate or malfunctioning terminal, or from an incorrect setting of switches on the terminal (for example, setting a terminal *caps lock* to transmit everything in uppercase). The SHARP APL system requires an APL colon as part of the sign-on. If you are communicating from a terminal without an APL character set, you may be transmitting the wrong character for colon. *Note:* To help clarify what is needed, the system ends its message to you with the **:** character it expects you to send between your user number and your password.

index error

Fr: *erreur d'indexation*
D: *indexfehler*

For a description of this message, see the entry in the preceding section under event 3.

indexfehler

E: *index error*
Fr: *erreur d'indexation*

For a description of this message, see the entry in the preceding section under event 3.

input interrupt

Fr: *interruption en entree*
D: *eingabe unterbrechung*

For a description of this message, see the entry in the preceding section under event 1004.

interface capacity exceeded

Fr: *vp: interface surchargee*
D: *gv kapazitaet ueberschritten*

For a description of this message, see the entry in the preceding section under event 74.

interface quota exhausted

Fr: *vp: quota epuise*
D: *gv limit erreicht*

For a description of this message, see the entry in the preceding section under event 72.

interrupt

Fr: *interruption*
D: *unterbrechung*

For a description of this message, see the entry in the preceding section under event 1003.

interruption

E: *interrupt*
D: *unterbrechung*

For a description of this message, see the entry in the preceding section under event 1003.

interruption en entree

E: *input interrupt*
D: *eingabe unterbrechung*

For a description of this message, see the entry in the preceding section under event 1004.

is...

Fr: *c'est...*
D: *ist...*

Several system commands permit you either to set or to inquire about a system attribute. For example, *)wsid x* sets the workspace name to *x* but *)wsid* alone reports but does not change the current name. When used to set an attribute, these commands report the former value, preceded by *was*. When used alone, they report the current value preceded by *is*.

ist...

E: *is...*
Fr: *c'est...*

For a description of this message, see the entry under the English-language version, *is...* .

kein platz

E: *no space*
Fr: *pas de place*

For a description of this message, see the entry under the English-language version, *no space*.

kennung bereits in verwendung

E: *identification in use*
Fr: *vp: identification deja utilisee*

For a description of this message, see the entry in the preceding section under event 77.

laengenfehler

E: *length error*
Fr: *erreur de longueur*

For a description of this message, see the entry in the preceding section under event 5.

length error

Fr: *erreur de longueur*
D: *laengenfehler*

For a description of this message, see the entry in the preceding section under event 5.

library table full

Fr: *debordement de table de bibliotheque*
D: *bibliothekstabelle voll*

The system's table of workspace names has no room to accommodate the name of the workspace you propose to save. Notify the system administrators so that corrective action can be taken. This event is not trappable.

ligne trop longue

E: *line too long*
D: *eingabe zu lang*

For a description of this message, see the entry under the English-language version, *line too long*.

line too long

Fr: *ligne trop longue*
D: *eingabe zu lang*

The ∇-editor cannot handle a line that exceeds 250 characters. You have attempted to edit such a line, or while editing, have indicated changes that would make the line exceed that length.

message lost

Fr: *message perdu*
D: *anfrage verloren*

The message you entered with the system command)*msg*,)*msgn*,)*opr*, or)*oprn* has not been sent because the system received an interrupt signal from your terminal after you sent the message but before it was dispatched to the recipient.

message perdu

E: *message lost*
D: *anfrage verloren*

For a description of this message, see the entry under the English-language version, *message lost*.

name ohne wert

E: *value error*
Fr: *objet inexistant*

For a description of this message, see the entry in the preceding section under event 6.

nicht ansprechbar

E: *incommunicado*
Fr: *non receptif*

For a description of this message, see the entry under the English-language version, *incommunicado*.

nicht gefunden:

Fr: *non trouve:*
E: *not found:*

For a description of this message, see the entry under the English-language version, *not found:*.

nicht geloescht:

E: *not erased:*
D: *non efface:*

For a description of this message, see the entry under the English-language version, *not erased:*.

nicht gespeichert, dieser ab heisst ...

E: *not saved, this ws is ...*
Fr: *non, c'est espace est ...*

For a description of this message, see the entry under the English-language version, *not saved... .*

nicht implementiert

E: *nonce error*
Fr: *enonce non defini*

For a description of this message, see the entry in the preceding section under event 16.

nicht kopiert:

E: *not copied:*
Fr: *non copie:*

For a description of this message, see the entry under the English-language version, *not copied:*.

no shares

Fr: *vp: non operationnel*
D: *gv prozessor nicht aktiv*

For a description of this message, see the entry in the preceding section under event 73.

no space

Fr: *pas de place*
D: *kein platz*

The system is unable to save the workspace as you requested because it lacks the necessary space. Notify the system administrators so that corrective action can be taken. This event is not trappable.

non, c'est espace est...

E: *not saved, this ws is ...*
D: *nicht gespeichert, dieser ab heisst ...*

For a description of this message, see the entry under the English-language version, *not saved... .*

nonce error

Fr: *enonce non defini*
D: *nicht implementiert*

For a description of this message, see the entry in the preceding section under event 16.

non copie:

E: *not copied:*
D: *nicht kopiert:*

For a description of this message, see the entry under the English-language version, *not copied:.*

non efface:

E: *not erased:*
D: *nicht geloescht:*

For a description of this message, see the entry under the English-language version, *not erased:.*

non receptif

E: *incommunicado*
D: *nicht ansprechbar*

For a description of this message, see the entry under the English-language version, *incommunicado.*

non trouve:

E: *not found:*
D: *nicht gefunden:*

For a description of this message, see the entry under the English-language version, *not found:.*

not copied:

Fr: *non copie:*
D: *nicht kopiert:*

The listed items have not been copied either because their names conflict with protected names in your active workspace, or because you have tried to copy a locked verb from a sealed workspace. A name in your active workspace may be protected because the active workspace is sealed; because it is the name of an active verb; or because the command you entered was *)pcopy* rather than *)copy*.

not erased:

Fr: *non efface:*
D: *nicht geloescht:*

The listed items have not been erased by your *)erase* command. An object is not erased if it is a locked verb in a sealed workspace, or an active verb.

not found:

E: *non trouve:*
D: *nicht gefunden:*

The listed names are not the names of global objects in the active workspace (if you tried to erase them) or in the source workspace (if you tried to copy them). Note that these commands apply only to global objects.

not saved, this ws is ...

Fr: *non, c'est espace est ...*
D: *nicht gespeichert, dieser ab heisst ...*

You have invoked the command *)save* in a workspace with no current name, or the command *)save* ***wsname*** when the workspace name you provided matches the name of an existing saved workspace, but does not match the current name of the active workspace. This event is not trappable.

number in use

E: *numero actif*
D: *benutzernummer in gebrauch*

Your attempt to sign on a T-task is unacceptable because a T-task on your user number is already signed on and you do not have authorization to start multiple T-tasks.

You may encounter this message because you simply forgot to sign off an earlier session. Sometimes it happens when you turn off power to a terminal without explicitly signing off; the SHARP APL system simply sees you as inactive. Certain communications link failures may produce similar symptoms.

If you are unable to sign off the earlier task, you may be able to get another user to start an N-task on your user number (which requires revealing your password) and then have the N-task bounce the T-task. Alternatively, you could attempt to contact the SHARP APL system operator by some other means and again request the operator to bounce your T-task. Notice that while connected but not signed on, with the command *)opr* you *can* send *one* message to the operator, to request assistance.

number locked out...

Fr: *autorisation retiree...*
D: *benutzernummer gesperrt...*

Although the number and password with which you have tried to sign on are valid, the user number has been locked out of the system. Consult the administrators of your SHARP APL system. With *)opr* you can send one message to the operator even when you are not signed on.

number not in...

Fr: *numero inconnu sur...*
D: *benutzernummer existiert nicht ...*

The combination of number and password supplied in your attempt to sign on is not valid. This may be because you have the number wrong, or because the password you supplied is not appropriate to that number. The system maintains a log of invalid attempts to sign on.

numero actif

E: *number in use*
D: *benutzernummer in gebrauch*

For a description of this message, see the entry under the English-language version, *number in use*.

numero inconnu sur...

E: *number not in ...*
D: *benutzernummer existiert nicht...*

For a description of this message, see the entry under the English-language version, *number not in... .*

objet inexistant

E: *value error*
D: *name ohne wert*

For a description of this message, see the entry in the preceding section under event 6.

pas de place

E: *ws full*
D: *ab voll*

For a description of this message, see the entry in the preceding section under event 1.

patience please

Fr: *patience please*
D: *patience please*

The system has not yet been able to terminate a T-task on your user number, even though it is identified as *bounced.* This may be a transitory condition. If it persists, contact the system administrators. (Note that you can send one message with the command *)opr* even when not signed on.)

processor table full

Fr: *vp: trop d'utilisateurs*
D: *prozessoren tabelle voll*

For a description of this message, see the entry in the preceding section under event 76.

prozessoren tabelle voll

E: *processor table full*
Fr: *vp: trop d'utilisateurs*

For a description of this message, see the entry in the preceding section under event 76.

□*xx errone*

E: □*xx error*
D: □*xx fehler*

For a description of this message, see the entry under the English-language version, □*xx error*.

□*xx error*

Fr: □*xx errone*
D: □*xx fehler*

You have invoked a verb requiring a valid value of the indicated system variable, but it is not present. For example, if you use indexing when □*io* has no valid value, you will see the message □*io error*; or if you attempt to display something when □*pw* has no valid value, you will see the message □*pw error*. Check the requirements for each system variable in the section devoted to it.

□*xx fehler*

E: □*xx error*
Fr: □*xx errone*

For a description of this message, see the entry under the English-language version, □*xx error*.

quota epuise

E: *ws quota used up*
D: *ab limit erreicht*

For a description of this message, see the entry under the English-language version, *ws quota used up*.

rangfehler

E: *rank error*
Fr: *erreur de rang*

For a description of this message, see the entry in the preceding section under event 4.

rank error

Fr: *erreur de rang*
D: *rangfehler*

For a description of this message, see the entry in the preceding section under event 4.

result error

Fr: *resultat inexistant*
D: *resultat fehlt*

For a description of this message, see the entry in the preceding section under event 8.

resultat fehlt

E: *result error*
Fr: *resultat inexistant*

For a description of this message, see the entry in the preceding section under event 8.

resultat inexistant

E: *result error*
D: *resultat fehlt*

For a description of this message, see the entry in the preceding section under event 8.

saved...

Fr: *copie du...*
D: *gespeichert...*

This message (followed by a timestamp) appears when you load or copy from a saved workspace. It indicates the date and time at which the workspace was saved.

sent

Fr: *transmis*
D: *gesendet*

A message that you entered with the command)*msg* or)*msgn* or)*opr* or)*oprn* has been dispatched to the recipient's terminal.

share table full

Fr: *vp: trop d'offres sur le systeme*
D: *gv tabelle voll*

For a description of this message, see the entry in the preceding section under event 75.

si error

Fr: *erreur de pile*
D: *si fehler*

For a description of this message, see the entry in the preceding section under event 9.

si fehler

E: *si error*
Fr: *erreur de pile*

For a description of this message, see the entry in the preceding section under event 9.

signed off

Fr: *usager absent*
D: *abgemeldet*

The task to which you have addressed a message via the *)msg* or *)msgn* command is not the number of a current task.

stop

E: *stop*
Fr: *stop*

For a description of this message, see the entry in the preceding section under event 1001.

sv interrupt

Fr: *vp: interruption*
D: *gv unterbrechung*

For a description of this message, see the entry in the preceding section under event 1005.

swap error, clear ws

Fr: *erreur d'echange, espace vide*
D: *swap fehler, ab leer*

For a description of this message, see the entry in the preceding section under event 67.

swap fehler, ab leer

E: *swap error, clear ws*
Fr: *erreur d'echange, espace vide*

For a description of this message, see the entry in the preceding section under event 67.

symbol table full

Fr: *debordement de table de symboles*
D: *symboltabelle voll*

For a description of this message, see the entry in the preceding section under event 15.

symboltabelle voll

E: *symbol table full*
Fr: *debordement de table de symboles*

For a description of this message, see the entry in the preceding section under event 15.

syntax error

Fr: *erreur de syntaxe*
D: *syntaxfehler*

For a description of this message, see the entry in the preceding section under event 2.

syntaxe de liaison invalide:

E: *incorrect signon:*
D: *fehlerhafte anmeldung:*

For a description of this message, see the entry under the English-language version, *incorrect signon:*.

syntax fehler

E: *syntax error*
Fr: *erreur de syntaxe*

For a description of this message, see the entry in the preceding section under event 2.

system error, clear ws

Fr: *erreur systeme, espace vide*
D: *system fehler, ab leer*

Following a system error, the system in effect executes the command *)clear* on your behalf. It does this because the workspace has been damaged and may contain spurious data that could be dangerous to the system. This event is not trappable.

For a description of this message, see the entry in the preceding section under event 68.

system fehler, ab leer

E: *system error, clear ws*
Fr: *erreur systeme, espace vide*

For a description of this message, see the entry in the preceding section under event 68.

systeme incapable d'ecrire cet espace

E: *ws not writable*
D: *ab nicht speicherbar*

For a description of this message, see the entry in the preceding section under event 95.

transmis

E: *sent*
D: *gesendet*

For a description of this message, see the entry under the English-language version, *sent*.

unerlaubter bibliothekszugriff

Fr: *bibliotheque inexistante*
E: *improper library reference*

For a description of this message, see the entry under the English-language version, *improper library reference*.

ungueltige systemanweisung

E: *incorrect command*
Fr: *commande invalide*

For a description of this message, see the entry under the English-language version, *incorrect command*.

unterbrechung

E: *interrupt*
D: *interruption*

For a description of this message, see the entry in the preceding section under event 1001–1007, and the discussion of event 1003.

unzulaessiges argument

E: *domain error*
Fr: *erreur de domaine*

For a description of this message, see the entry in the preceding section under event 11.

unzulaessiges zeichen in der eingabe

E: *entry error*
Fr: *erreur en entree*

For a description of this message, see the entry under the English-language version, *entry error*.

usager absent

E: *signed off*
D: *abgemeldet*

For a description of this message, see the entry under the English-language version, *signed off*.

value error

Fr: *objet inexistant*
D: *name ohne wert*

For a description of this message, see the entry in the preceding section under event 6.

vp: identification deja utilisee

E: *identification in use*
D: *kennung bereits in verwendung*

For a description of this message, see the entry in the preceding section under event 77.

vp: interface surchargee

E: *interface capacity exceeded*
D: *gv kapazitaet ueberschritten*

For a description of this message, see the entry in the preceding section under event 74.

vp: interruption

E: *sv interrupt*
D: *gv unterbrechung*

For a description of this message, see the entry in the preceding section under event 1005.

vp: non operationnel

E: *no shares*
D: *gv prozessor nicht aktiv*

For a description of this message, see the entry in the preceding section under event 73.

vp: quota epuise

E: *interface quota exhausted*
D: *gv limit erreicht*

For a description of this message, see the entry in the preceding section under event 72.

vp: trop d'offres sur le systeme

E: *share table full*
D: *gv tabelle voll*

For a description of this message, see the entry in the preceding section under event 75.

vp: trop d'utilisateurs

E: *processor table full*
D: *prozessoren tabelle voll*

For a description of this message, see the entry in the preceding section under event 76.

war...

E: *was...*
Fr: *c'etait...*

For a description of this message, see the entry under the English-language version, *was...* .

was...

Fr: *c'etait...*
D: *war...*

Following several commands that change the current state of something, the acknowledgment is a report of the former state, preceded by this word. (See also the discussion of *is...* , earlier in this section.)

`ws full`

Fr: *pas de place*
D: *ab voll*

For a description of this message, see the entry in the preceding section under event 1.

`ws locked`

Fr: *espace verrouille*
D: *ab gesperrt*

For a description of this message, see the entry in the preceding section under event 45.

`ws not found`

Fr: *espace inexistant*
D: *ab nicht gefunden*

For a description of this message, see the entry in the preceding section under event 46.

`ws not readable`

Fr: *espace illisible*
D: *ab nicht lesbar*

For a description of this message, see the entry in the preceding section under event 47.

`ws not writable`

Fr: *systeme incapable d'ecrire cet espace*
D: *ab nicht speicherbar*

For a description of this message, see the entry in the preceding section under event 95.

`ws quota used up`

Fr: *quota epuise*
D: *ab limit erreicht*

You have tried to save a workspace, but already have saved your full quota of saved workspaces. (Each user is entitled to have a workspace called `continue` saved following a system crash, bounce, or a communications failure. The `continue` workspace does not count against your quota, and you should not use it for regular storage because it could be replaced by the active workspace at any instant, due to a future crash, bounce, or communications failure.)

Remedy: Drop an unneeded workspace, or request a larger workspace quota from the system administrators.